PCT

INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(51) International Patent Classification 6 : G06F 9/44, 1/00	**A1**	(11) International Publication Number: **WO 99/01815** (43) International Publication Date: 14 January 1999 (14.01.99)

(21) International Application Number: PCT/US98/12017

(22) International Filing Date: 9 June 1998 (09.06.98)

(30) Priority Data:
328057   9 June 1997 (09.06.97)   NZ

(71) Applicant (for all designated States except US): INTERTRUST, INCORPORATED [US/US]; 460 Oakmead Parkway, Sunnyvale, CA 94086 (US).

(72) Inventors; and
(75) Inventors/Applicants (for US only): COLLBERG, Christian, Sven [SE/NZ]; 25 Glenalmond Road, Mt. Eden, Auckland (NZ). THOMBORSON, Clark, David [US/NZ]; 3/61 Fancourt Street, Meadowbanks, Auckland (NZ). LOW, Douglas, Wai, Kok [NZ/NZ]; 56 Almorah Road, Epsom, Auckland 3 (NZ).

(74) Agents: OGONOWSKY, Brian, D. et al.; Skjerven, Morrill, MacPherson, Franklin & Friel LLP, Suite 700, 25 Metro Drive, San Jose, CA 95110 (US).

(81) Designated States: AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, CA, CH, CN, CU, CZ, DE, DK, EE, ES, FI, GB, GE, GH, GM, GW, HU, ID, IL, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MD, MG, MK, MN, MW, MX, NO, NZ, PL, PT, RO, RU, SD, SE, SG, SI, SK, SL, TJ, TM, TR, TT, UA, UG, US, UZ, VN, YU, ZW, ARIPO patent (GH, GM, KE, LS, MW, SD, SZ, UG, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, CH, CY, DE, DK, ES, FI, FR, GB, GR, IE, IT, LU, MC, NL, PT, SE), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, ML, MR, NE, SN, TD, TG).

Published
With international search report.

---

(54) Title: OBFUSCATION TECHNIQUES FOR ENHANCING SOFTWARE SECURITY

(57) Abstract

The present invention provides obfuscation techniques for enhancing software security. In one embodiment, a method for obfuscation techniques for enhancing software security includes selecting a subset of code (e.g., compiled source code of an application) to obfuscate, and obfuscating the selected subset of the code. The obfuscating includes applying an obfuscating transformation to the selected subset of the code. The transformed code can be weakly equivalent to the untransformed code. The applied transformation can be selected based on a desired level of security (e.g., resistance to reverse engineering). The applied transformation can include a control transformation that can be creating using opaque constructs, which can be constructed using aliasing and concurrency techniques. Accordingly, the code can be obfuscated for enhanced software security based on a desired level of obfuscation (e.g., based on a desired potency, resilience, and cost).

OBFUSCATION TECHNIQUES FOR ENHANCING SOFTWARE SECURITY

5   FIELD OF THE INVENTION

The present invention relates to methods and apparatus for preventing, or at least hampering, interpretation, decoding, or reverse engineering of software. More particularly, although not exclusively,

10   the present invention relates to methods and apparatus for increasing the structural and logical complexity of software by inserting, removing, or rearranging identifiable structure or information from the software in such a way as to exacerbate the difficulty of the

15   process of decompilation or reverse engineering.

BACKGROUND

The nature of software renders it susceptible to analysis and copying by third parties. There have been

20   considerable efforts to enhance software security, which have met with mixed success. Such security concerns relate to the need to prevent unauthorized copying of software and a desire to conceal programming techniques in which such techniques can be determined

25   via reverse engineering.

Established legal avenues, such as copyright, provide a measure of legislative protection. However, enforcing legal rights created under such regimes can be expensive and time consuming. Further, the

30   protection afforded to software under copyright does

SUBSTITUTE SHEET (RULE 26)

not cover programming techniques. Such techniques
(i.e., the function as opposed to the form of the
software) are legally difficult to protect. A reverse
engineer could escape infringement by rewriting the
5   relevant software, ab initio, based on a detailed
knowledge of the function of the software in question.
Such knowledge can be derived from analyzing the data
structures, abstractions, and organization of the code.

Software patents provide more comprehensive
10   protection. However, it is clearly an advantage to
couple legal protection of software with technical
protection.

Previous approaches to the protection of
proprietary software have either used encryption-based
15   hardware solutions or have been based on simple
rearrangements of the source code structure. Hardware-
based techniques are non-ideal in that they are
generally expensive and are tied to a specific platform
or hardware add-on. Software solutions typically
20   include trivial code obfuscators, such as the Crema
obfuscator for Java™. Some obfuscators target the
lexical structure of the application and typically
remove source code formatting and comments and rename
variables. However, such an obfuscation technique does
25   not provide sufficient protection against malicious
reverse engineering: reverse engineering is a problem
regardless of the form in which the software is
distributed. Further, the problem is exacerbated when
the software is distributed in hardware-independent
30   formats that retain much or all of the information in

-2-

SUBSTITUTE SHEET (RULE 26)

the original source code. Examples of such formats are
Java™ bytecode and the Architecture Neutral
Distribution Format (ANDF).

5   Software development can represent a significant
investment in time, effort, and skill by a programmer.
In the commercial context, the ability to prevent a
competitor from copying proprietary techniques can be
critical.

10   SUMMARY

The present invention provides methods and
apparatus for obfuscation techniques for software
security, such as computer implemented methods for
reducing the susceptibility of software to reverse
15   engineering (or to provide the public with a useful
choice). In one embodiment, a computer implemented
method for obfuscating code, includes testing for
completion of supplying one or more obfuscation
transformations to the code, selecting a subset of the
20   code to obfuscate, selecting an obfuscating transform
to apply, applying the transformation, and returning to
the completion testing step.

In an alternative embodiment, the present
invention relates to a method of controlling a computer
25   so that software running on, stored on, or manipulated
by the computer exhibits a predetermined and controlled
degree of resistance to reverse engineering, including
applying selected obfuscating transformations to
selected parts of the software, in which a level of
30   obfuscation is achieved using a selected obfuscation

-3-

transformation so as to provide a required degree of
resistance to reverse engineering, effectiveness in
operation of the software and size of transformed
software, and updating the software to reflect the
5 obfuscating transformations.

In a preferred embodiment, the present invention
provides a computer implemented method for enhancing
software security, including identifying one or more
source code input files corresponding to the source
10 software for the application to be processed, selecting
a required level of obfuscation (e.g., potency),
selecting a maximum execution time or space penalty
(e.g., cost), reading and parsing the input files,
optionally along with any library or supplemental files
15 read directly or indirectly by the source code,
providing information identifying data types, data
structures, and control structures used by the
application to be processed, and constructing
appropriate tables to store this information,
20 preprocessing information about the application, in
response to the preprocessing step, selecting and
applying obfuscating code transformations to source
code objects, repeating the obfuscating code
transformation step until the required potency has been
25 achieved or the maximum cost has been exceeded, and
outputting the transformed software.

Preferably, the information about the application
is obtained using various static analysis techniques
and dynamic analysis techniques. The static analysis
30 techniques include inter-procedural dataflow analysis

-4-

SUBSTITUTE SHEET (RULE 26)

and data dependence analysis. The dynamic analysis
techniques include profiling, and optionally,
information can be obtained via a user. Profiling can
be used to determine the level of obfuscation, which
5   can be applied to a particular source code object.
Transformations can include control transformations
created using opaque constructs in which an opaque
construct is any mathematical object that is
inexpensive to execute from a performance standpoint,
10   simple for an obfuscator to construct, and expensive
for a deobfuscator to break. Preferably, opaque
constructs can be constructed using aliasing and
concurrency techniques. Information about the source
application can also be obtained using pragmatic
15   analysis, which determines the nature of language
constructs and programming idioms the application
contains.

The potency of an obfuscation transformation can
be evaluated using software complexity metrics.
20   Obfuscation code transformations can be applied to any
language constructs: for example, modules, classes, or
subroutines can be split or merged; new control and
data structures can be created; and original control
and data structures can be modified. Preferably, the
25   new constructs added to the transformed application are
selected to be as similar as possible to those in the
source application, based on the pragmatic information
gathered during preprocessing. The method can produce
subsidiary files including information about which
30   obfuscating transformations have been applied and

-5-

information relating obfuscated code of the transformed application to the source software.

Preferably, the obfuscation transformations are selected to preserve the observable behavior of the
5   software such that if P is the untransformed software, and P' is the transformed software, P and P' have the same observable behavior. More particularly, if P fails to terminate or terminates with an error condition, then P' may or may not terminate, otherwise
10 P' terminates and produce the same output as P. Observable behavior includes effects experienced by a user, but P and P' may run with different detailed behavior unobservable by a user. For example, detailed behavior of P and P' that can be different includes
15 file creation, memory usage, and network communication.

In one embodiment, the present invention also provides a deobfuscating tool adopted to remove obfuscations from an obfuscated application by use of slicing, partial evaluation, dataflow analysis, or
20 statistical analysis.

BRIEF DESCRIPTION OF THE DRAWINGS

The present invention will now be described by way of example only and with reference to the drawings in
25 which:

FIG. 1 illustrates a data processing system in accordance with the teachings of the present invention;

FIG. 2 illustrates a classification of software protection including categories of obfuscating
30 transformations;

-6-

FIGs. 3a and 3b show techniques for providing software security by (a) server-side execution and (b) partial server-side execution;

FIGs. 4a and 4b show techniques for providing

5   software security by (a) using encryption and (b) using signed native code;

FIG. 5 shows a technique for providing software security through obfuscation;

FIG. 6 illustrates the architecture of an example

10 of an obfuscating tool suitable for use with Java™ applications;

FIG. 7 is a table that tabulates a selection of known software complexity metrics;

FIGs. 8a and 8b illustrate the resilience of an

15 obfuscating transformation;

FIG. 9 shows different types of opaque predicates;

FIGs. 10a and 10b provide examples of (a) trivial opaque constructs and (b) weak opaque constructs;

FIG. 11 illustrates an example of a computation

20 transformation (branch insertion transformation);

FIGs. 12a through 12d illustrate a loop condition insertion transformation;

FIG. 13 illustrates a transformation that transforms reducible flowgraphs into non-reducible

25 flowgraphs;

FIG. 14 shows that a section of code can be parallelized if it contains no data dependencies;

FIG. 15 shows that a section of code that contains no data dependencies can be split into concurrent

threads by inserting appropriate synchronization
primitives;

     FIG. 16 shows how procedures P and Q are inlined
at their call-sites and then removed from the code;

5   FIG. 17 illustrates inlining method calls;

     FIG. 18 shows a technique for interleaving two
methods declared in the same class;

     FIG. 19 shows a technique for creating several
different versions of a method by applying different
10   sets of obfuscating transformations to the original
code;

     FIGs. 20a through 20c provide examples of loop
transformations including (a) loop blocking, (b) loop
unrolling, and (c) loop fission;

15   FIG. 21 shows a variable splitting example;

     FIG. 22 provides a function constructed to
obfuscate strings "AAA", "BAAAA", and "CCB";

     FIG. 23 shows an example merging two 32-bit
variables x and y into one 64-bit variable Z;

20   FIG. 24 illustrates an example of a data
transformation for array restructuring;

     FIG. 25 illustrates modifications of an
inheritance hierarchy;

     FIG. 26 illustrates opaque predicates constructed
25   from objects and aliases;

     FIG. 27 provides an example of opaque constructs
using threads;

     FIGs. 28a through 28d illustrate obfuscation vs.
deobfuscation in which (a) shows an original program
30   including three statements, S_{1-3}, being obfuscated, (b)

-8-

shows a deobfuscator identifying "constant" opaque
predicates, (c) shows the deobfuscator determining the
common code in the statements, and (d) shows the
deobfuscator applying some final simplifications and
5   returning the program to its original form;

     FIG. 29 shows an architecture of a Java^{TM}
deobfuscation tool;

     FIG. 30 shows an example of statistical analysis
used for evaluation;

10   FIGs. 31a and 31b provide tables of an overview of
various obfuscating transforms; and

     FIG. 32 provides an overview of various opaque
constructs.

15   DETAILED DESCRIPTION

     The following description will be provided in the
context of a Java^{TM} obfuscation tool, which is currently
being developed by the applicants. However, it will be
apparent to one of ordinary skill in the art that the
20   present techniques are applicable to other programming
languages and the invention is not to be construed as
restricted to Java^{TM} applications. The implementation
of the present invention in the context of other
programming languages is considered to be within the
25   purview of one of ordinary skill in the art. The
exemplary embodiment that follows is, for clarity,
specifically targeted at a Java^{TM} obfuscating tool.

     In the description below, the following
nomenclature will be used. P is the input application
30   to be obfuscated; P' is the transformed application; T

-9-

is a transformation such that T transforms P into P'.
P(T)P' is an obfuscating transformation if P and P'
have the same observable behavior. Observable behavior
is defined generally as behavior experienced by the
5   user. Thus, P' may have side effects such as creating
files that P does not, so long as these side effects
are not experienced by the user. P and P' do not need
to be equally efficient.

10   Exemplary Hardware
       FIG. 1 illustrates a data processing system in
accordance with the teachings of the present invention.
FIG. 1 shows a computer 100, which includes three major
elements. Computer 100 includes an input/output (I/O)
15   circuit 120, which is used to communicate information
in appropriately structured form to and from other
portions of computer 100. Computer 100 includes a
control processing unit (CPU) 130 in communication with
I/O circuit 120 and a memory 140 (e.g., volatile and
20   non-volatile memory). These elements are those
typically found in most general purpose computers and,
in fact, computer 100 is intended to be representative
of a broad category of data processing devices. A
raster display monitor 160 is shown in communication
25   with I/O circuit 120 and issued to display images
generated by CPU 130. Any well known variety of
cathode ray tube (CRT) or other type of display can be
used as display 160. A conventional keyboard 150 is
also shown in communication with I/O 120. It will be
30   appreciated by one of ordinary skill in the art that

                              -10-

computer 100 can be part of a larger system. For
example, computer 100 can also be in communication with
a network (e.g., connected to a local area network
(LAN)).

5   In particular, computer 100 can include
obfuscating circuitry for enhancing software security
in accordance with the teachings of the present
invention, or as will be appreciated by one of ordinary
skill in the art, the present invention can be

10 implemented in software executed by computer 100 (e.g.,
the software can be stored in memory 140 and executed
on CPU 130). For example, an unobfuscated program P
(e.g., an application), stored in memory 140, can be
obfuscated by an obfuscator executing on CPU 130 to

15 provide an obfuscated program P', stored in memory 140,
in accordance with one embodiment of the present
invention.

        Overview of the Detailed Description

20   FIG. 6 shows the architecture of a Java^{TM}
obfuscator. According to the inventive method, Java^{TM}
application class files are passed along with any
library files. An inheritance tree is constructed as
well as a symbol table, providing type information for

25 all symbols and control flow graphs for all methods.
The user may optionally provide profiling data files as
generated by Java^{TM} profiling tools. This information
can be used to guide the obfuscator to ensure that
frequently executed parts of the application are not

30 obfuscated by very expensive transformations.

-11-

Information is gathered about the application using standard compiler techniques such as interprocedural dataflow analysis and data dependence analysis. Some can be provided by the user and some by specialized

5   techniques. The information is used to select and apply the appropriate code transformations.

Appropriate transformations are selected. The governing criteria used in selecting the most suitable transformation include the requirement that the chosen

10 transformation blend in naturally with the rest of the code. This can be dealt with by favoring transformations with a high appropriateness value. A further requirement is that transformations which yield a high level of obfuscation with low execution time

15 penalty should be favored. This latter point is accomplished by selecting transformations that maximize potency and resilience, and minimize cost.

An obfuscation priority is allocated to a source code object. This will reflect how important it is to

20 obfuscate the contents of the source code object. For example, if a particular source code object contains highly sensitive proprietary material, then the obfuscation priority will be high. An execution time rank is determined for each method, which equals 1 if

25 more time is spent executing the method than any other.

The application is then obfuscated by building the appropriate internal data structures, the mapping from each source code object to the appropriate transformation, the obfuscation priority, and the

30 execution time rank. The obfuscating transformations

-12-

are applied until the required level of obfuscation has

been achieved or until the maximum execution time

penalty is exceeded. The transformed application is

then written.

5   The output of the obfuscation tool is a new

application that is functionally equivalent to the

original. The tool can also produce Java™ source files

annotated with information about which transformations

have been applied and how the obfuscated code relates

10   to the original application.

A number of examples of obfuscating

transformations will now be described, again in the

context of a Java™ obfuscator.

Obfuscating transformations can be evaluated and

15   classified according to their quality. The quality of

a transformation can be expressed according to its

potency, resilience, and cost. The potency of a

transformation is related to how obscure P' is in

relation to P. Any such metric will be relatively

20   vague as it necessarily depends on human cognitive

abilities. For the present purposes it is sufficient

to consider the potency of a transformation as a

measure of the usefulness of the transformation. The

resilience of a transformation measures how well a

25   transformation holds up to an attack from an automatic

deobfuscator. This is a combination of two factors:

programmer effort and deobfuscator effort. Resilience

can be measured on a scale from trivial to one-way.

One-way transformations are extreme in that they cannot

30   be reversed. The third component is transformation

-13-

SUBSTITUTE SHEET (RULE 26)

execution cost. This is the execution time or space penalty incurred as a result of using the transformed application P'. Further details of transformation evaluation are discussed below in the detailed

5   description of the preferred embodiments. The main classification of obfuscating transformations is shown in FIG. 2c with details given in FIGs. 2e through 2g.

Examples of obfuscating transforms are as follows: Obfuscating transforms may be categorized as follows:

10   control obfuscation, data obfuscations, layout obfuscations, and preventive obfuscations. Some examples of these are discussed below.

Control obfuscations include aggregation transformations, ordering transformations, and

15   computation transformations.

Computation transformations include: concealing real control flow behind irrelevant non-functional statements; introducing code sequences at the object code level for which there exist no corresponding high-

20   level language constructs; and removing real control flow abstractions or introducing spurious ones.

Considering the first classification (control flow), the Cyclomatic and Nesting complexity metrics suggest that there is a strong correlation between the

25   perceived complexity of a piece of code and the number of predicates it contains. Opaque predicates enable the construction of transformations which introduce new predicates into the program.

Referring to FIG. 11a, an opaque predicate P^T is

30   inserted into the basic block S where $S = S_1 \ldots S_n$. This

-14-

splits S in half. The P^T predicate is irrelevant code, because it will always evaluate to True. In FIG. 11b, S is again broken into two halves, which are transformed into two different obfuscated versions S^a

5   and S^b. Therefore, it will not be obvious to a reverse engineer that S^a and S^b perform the same function. FIG. 11c is similar to FIG. 11b, however, a bug is introduced into S^b. The P^T predicate always selects the correct version of the code, S^a.

10   Another type of obfuscation transformation is a data transformation. An example of a data transformation is deconstructing arrays to increase the complexity of code. An array can be split into several subarrays, two or more arrays can be merged into a

15   single array, or the dimensions of an array can be increased (flattening) or decreased (folding). FIG. 24 illustrates a number of examples of array transformations. In statements (1-2), an array A is split into two subarrays A1 and A2. A1 contains

20   elements with even indices and A2 contains elements with odd indices. Statements (3-4) illustrate how two integer arrays B and C can be interleaved to produce an array BC. The elements from B and C are evenly spread throughout the transformed array. Statements (6-7)

25   illustrate folding of array D into array D1. Such transformations introduce previously absent data structure or remove existing data structure. This can greatly increase the obscurity of the program as, for example, in declaring a 2-dimensional array a

30   programmer usually does so for a purpose, with the

chosen structure mapping onto the corresponding data.
If that array is folded into a 1-d structure, a reverse
engineer would be deprived of valuable pragmatic
information.

5   Another example of an obfuscating transformation
is a preventive transformation. In contrast to control
or data transformations, the main goal of preventive
transformations is not to obscure the program to a
human reader, but to make known automatic deobfuscation
10   techniques more difficult or to exploit known problems
in current deobfuscators or decompilers. Such
transformations are known as inherent and targeted,
respectively. An example of an inherent preventive
transformation is reordering a for-loop to run
15   backward. Such reordering is possible if the loop has
no loop-carried data dependencies. A deobfuscator
could perform the same analysis and reorder the loop to
forward execution. However, if a bogus data dependency
is added to the reversed loop, the identification of
20   the loop and its reordering would be prevented.
          Further specific examples of obfuscating
transformations are discussed below in the detailed
description of the preferred embodiments.

25   Detailed Description of the Preferred Embodiments
              It has become more and more common to
distribute software in forms that retain most or
all of the information present in the original
source code. An important example is Java
30   bytecode. Because such codes are easy to

-16-

decompile, they increase the risk of malicious reverse engineering attacks.

Accordingly, several techniques for technical protection of software secrets are provided in accordance with one embodiment of the present invention. In the detailed description of the preferred embodiments, we will argue that automatic code obfuscation is currently the most viable method for preventing reverse engineering. We then describe the design of a code obfuscator, an obfuscation tool that converts a program into an equivalent one that is more difficult to understand and reverse engineer.

The obfuscator is based on the application of code transformations, in many cases similar to those used by compiler optimizers. We describe a large number of such transformations, classify them, and evaluate them with respect to their potency (e.g., To what degree is a human reader confused?), resilience (e.g., How well are automatic deobfuscation attacks resisted?), and cost (e.g., How much performance overhead is added to the application?).

We finally describe various deobfuscation techniques (such as program slicing) and possible countermeasures an obfuscator could employ against them.

1 Introduction

SUBSTITUTE SHEET (RULE 26)

Given enough time, effort, and determination,
a competent programmer will always be able to
reverse engineer any application. Having gained
physical access to the application, the reverse
5   engineer can decompile it (using disassemblers or
decompilers) and then analyze its data structures
and control flow. This can either be done manually
or with the aid of reverse engineering tools, such
as program slicers.
10   Reverse engineering is not a new problem.
Until recently, however, it is a problem that has
received relatively little attention from software
developers, because most programs are large,
monolithic, and shipped as stripped, native code,
15   making them difficult (although never impossible)
to reverse engineer.
However, this situation is changing as it is
becoming more and more common to distribute
software in forms that are easy to decompile and
20   reverse engineer. Important examples include Java
bytecode and the Architecture Neutral Distribution
Format (ANDF). Java applications in particular
pose a problem to software developers. They are
distributed over the Internet as Java class files,
25   a hardware-independent virtual machine code that
retains virtually all the information of the
original Java source. Hence, these class files are
easy to decompile. Moreover, because much of the
computation takes place in standard libraries,

Java programs are often small in size and

therefore relatively easy to reverse engineer.

        The main concern of Java developers is not

outright reengineering of entire applications.

5   There is relatively little value in such behavior,

because it clearly violates copyright law [29] and

can be handled through litigation. Rather,

developers are mostly frightened by the prospect

of a competitor being able to extract proprietary

10   algorithms and data structures from their

applications in order to incorporate them into

their own programs. Not only does it give the

competitor a commercial edge (by cutting

development time and cost), but it is also

15   difficult to detect and pursue legally. The

last point is particularly valid for small

developers who may ill afford lengthy legal

battles against powerful corporations [22] with

unlimited legal budgets.

20   An overview of various forms of protection

for providing legal protection or security for

software is provided in FIG. 2. FIG. 2 provides a

classification of (a) kinds of protection against

malicious reverse engineering, (b) the quality of

25   an obfuscating transformation, (c) information

targeted by an obfuscating transformation, (d)

layout obfuscations, (e) data obfuscations, (f)

control obfuscations, and (g) preventive

obfuscations.

The various forms of technical protection of
intellectual property, which are available to
software developers are discussed below. We will
restrict our discussion to Java programs

5   distributed over the Internet as Java class-files,
although most of our results will apply to other
languages and architecture-neutral formats as
well, as will be apparent to one of ordinary skill
in the art. We will argue that the only

10   reasonable approach to the protection of mobile
code is code obfuscation. We will furthermore
present a number of obfuscating transformations,
classify them according to effectiveness and
efficiency, and show how they can be put to use in

15   an automatic obfuscation tool.

The remainder of the detailed description of
the preferred embodiments is structured as
follows. In Section 2, we give an overview of
different forms of technical protection against

20   software theft and argue that code obfuscation
currently affords the most economical prevention.
In Section 3, we give a brief overview of the
design of Kava, a code obfuscator for Java, which
is currently under construction. Sections 4 and 5

25   describe the criteria we use to classify and
evaluate different types of obfuscating
transformations. Sections 6, 7, 8, and 9 present
a catalogue of obfuscating transformations. In
Section 10, we give more detailed obfuscation

30   algorithms. In Section 11, we conclude with a

SUBSTITUTE SHEET (RULE 26)

summary of our results and a discussion of future
directions of code obfuscation.

2   Protecting Intellectual Property

5   Consider the following scenario. Alice is a
small software developer who wants to make her
applications available to users over the Internet,
presumably at a charge. Bob is a rival developer
who feels that he could gain a commercial edge
10   over Alice if he had access to her application's
key algorithms and data structures.
         This can be seen as a two-player game between
two adversaries: the software developer (Alice)
who tries to protect her code from attack, and the
15   reverse engineer (Bob) whose task it is to analyze
the application and convert it into a form that is
easy to read and understand. Note that it is not
necessary for Bob to convert the application back
to something close to Alice's original source; all
20   that is necessary is that the reverse engineered
code be understandable by Bob and his programmers.
Note also that it may not be necessary for Alice
to protect her entire application from Bob; it
probably consists mostly of "bread-and-butter
25   code" that is of no real interest to a competitor.
         Alice can protect her code from Bob's attack
using either legal or technical protection, such
as shown in FIG. 2a, which is discussed above.
While copyright law does cover software artifacts,
30   economic realities make it difficult for a small

-21-

company like Alice's to enforce the law against a
larger and more powerful competitor. A more
attractive solution is for Alice to protect her
code by making reverse engineering so technically
5   difficult that it becomes impossible or at the
very least economically inviable. Some early
attempts at technical protection are described by
Gosler. (James R. Gosler. Software protection:
Myth or reality? In CRYPTO'85 --- Advances in
10   Cryptology, pages 140--157, August 1985).

The most secure approach is for Alice not to
sell her application at all, but rather sell its
services. In other words, users never gain access
to the application itself but rather connect to
15   Alice's site to run the program remotely as shown
in FIG. 3a, paying a small amount of electronic
money every time. The advantage to Alice is that
Bob will never gain physical access to the
application and hence will not be able to reverse
20   engineer it. The downside is of course that, due
to limits on network bandwidth and latency, the
application may perform much worse than if it had
run locally on the user's site. A partial
solution is to break the application into two
25   parts: a public part that runs locally on the
user's site, and a private part (that contains the
algorithms that Alice wants to protect) that is
run remotely, for example, as shown in FIG. 3b.

Another approach would be for Alice to
30   encrypt her code before it is sent off to the

-22-

SUBSTITUTE SHEET (RULE 26)

users, for example, as shown in FIG. 4a.
Unfortunately, this only works if the entire
decryption/execution process takes place in
hardware.   Such systems are described in Herzberg
5   (Amir Herzberg and Shlomit S. Pinter. Public
protection of software. ACM Transactions on
Computer Systems, 5(4):371--393, November 1987.)
and Wilhelm (Uwe G. Wilhelm. Cryptographically
protected objects.
10   http://lsewww.epfl.ch/~wilhelm/CryPO. html, 1997).
If the code is executed in software by a virtual
machine interpreter (as is most often the case
with Java bytecodes), then it will always be
possible for Bob to intercept and decompile the
15   decrypted code.

     The Java™ programming language has gained
popularity mainly because of its architecture
neutral bytecode.   While this clearly facilitates
mobile code, it does decrease the performance by
20   an order of magnitude in comparison to native
code.   Predictably, this has lead to the
development of just-in-time compilers that
translate Java bytecodes to native code
on-the-fly.   Alice could make use of such
25   translators to create native code versions of her
application for all popular architectures.   When
downloading the application, the user's site would
have to identify the architecture/operating system
combination it is running, and the corresponding
30   version would be transmitted, for example, as

-23-

SUBSTITUTE SHEET (RULE 26)

shown in FIG. 4b. Only having access to the
native code will make Bob's task more difficult,
although not impossible. There is a further
complication with transmitting

5   native code. The problem is that --- unlike Java
bytecodes, which are subjected to bytecode
verification before execution, --- native codes
cannot be run with complete security on the user's
machine. If Alice is a trusted member of the

10   community, the user may accept her assurances that
the application does not do anything harmful at
the user's end. To make sure that no one tries to
contaminate the application, Alice would have to
digitally sign the codes as they are being

15   transmitted, to prove to the user that the code
was the original one written by her.

     The final approach we are going to consider
is code obfuscation, for example, as shown in FIG.
5. The basic idea is for Alice to run her

20   application through an obfuscator, a program that
transforms the application into one that is
functionally identical to the original but which
is much more difficult for Bob to understand. It
is our belief that obfuscation is a viable

25   technique for protecting software trade secrets
that has yet to receive the attention that it
deserves.

     Unlike server-side execution, code
obfuscation can never completely protect an

30   application from malicious reverse engineering

-24-

efforts. Given enough time and determination, Bob
will always be able to dissect Alice's application
to retrieve its important algorithms and data
structures. To aid this effort, Bob may try to run
5   the obfuscated code through an automatic
deobfuscator that attempts to undo the obfuscating
transformations.

Hence, the level of security from reverse
engineering that an obfuscator adds to an
10 application depends on, for example, (a) the
sophistication of the transformations employed by
the obfuscator, (b) the power of the available
deobfuscation algorithms, and (c) the amount of
resources (time and space) available to the
15 deobfuscator. Ideally, we would like to mimic the
situation in current public-key cryptosystems, in
which there is a dramatic difference in the cost
of encryption (finding large primes is easy) and
decryption (factoring large numbers is difficult).
20 We will see that there are, in fact, obfuscating
transformations that can be applied in polynomial
time but which require exponential time to
deobfuscate, as discussed below.

25 3. The Design of a Java Obfuscator
FIG. 6 shows an architecture of Kava, the
Java obfuscator. The main input to the tool is a
set of Java class files and the obfuscation level
required by the user. The user may optionally
30 provide files of profiling data, as generated by

Java profiling tools. This information can be
used to guide the obfuscator to make sure that
frequently executed parts of the application are
not obfuscated by very expensive transformations.
5   Input to the tool is a Java application, given as
a set of Java class files. The user also selects
the required level of obfuscation (e.g., potency)
and the maximum execution time/space penalty that
the obfuscator is allowed to add to the
10   application (the cost). Kava reads and parses the
class files along with any library files
referenced directly or indirectly. A complete
inheritance tree is constructed, as well as a
symbol table giving type information for all
15   symbols, and control flow graphs for all methods.

Kava contains a large pool of code
transformations, which are described below.
Before these can be applied, however, a
preprocessing pass collects various types of
20   information about the application in accordance
with one embodiment. Some kinds of information
can be gathered using standard compiler techniques
such as inter-procedural dataflow analysis and
data dependence analysis, some can be provided by
25   the user, and some are gathered using specialized
techniques. Pragmatic analysis, for example,
analyzes the application to see what sort of
language constructs and programming idioms it
contains.

-26-

The information gathered during the
preprocessing pass is used to select and apply
appropriate code transformations. All types of
language constructs in the application can be the
5   subject of obfuscation: for example, classes can
be split or merged, methods can be changed or
created, new control and data structures can be
created and original ones modified. New constructs
added to the application can be selected to be as
10   similar as possible to the ones in the source
application, based on the pragmatic information
gathered during the preprocessing pass.

The transformation process is repeated until
the required potency has been achieved or the
15   maximum cost has been exceeded. The output of the
tool is a new application -- functionally
equivalent to the original one -- normally given
as a set of Java class files. The tool will also
be able to produce Java source files annotated
20   with information about which transformations have
been applied, and how the obfuscated code relates
to the original source. The annotated source will
be useful for debugging.

25   4 Classifying Obfuscating Transformations

In the remainder of this detailed description
of the preferred embodiments we will describe,
classify, and evaluate various obfuscating
transformations. We start by formalizing the
30   notion of an obfuscating transformation:

-27-

SUBSTITUTE SHEET (RULE 26)

Definition 1 (Obfuscating Transformation)

Let P $\xrightarrow{T}$ P' be a legal obfuscating transformation
in which the following conditions must hold:

5

- If P fails to terminate or terminates with
an error condition, then P' may or may not
terminate.

10   - Otherwise, P' must terminate and produce
the same output as P.

Observable behavior is defined loosely as
"behavior as experienced by the user." This means
15   that P' may have side-effects (such as creating
files or sending messages over the Internet) that
P does not, as long as these side effects are not
experienced by the user. Note that we do not
require P and P' to be equally efficient. In fact,
20   many of our transformations will result in P'
being slower or using more memory than P .
The main dividing line between different
classes of obfuscation techniques is shown in FIG.
2c. We primarily classify an obfuscating
25   transformation according to the kind of
information it targets. Some simple
transformations target the lexical structure (the
layout) of the application, such as source code
formatting, and names of variables. In one
30   embodiment, the more sophisticated transformations

-28-

that we are interested in, target either the data
structures used by the application or its flow of
control.

Secondly, we classify a transformation
5   according to the kind of operation it performs on
the targeted information. As can be seen from
FIGs. 2d through 2g, there are several
transformations that manipulate the aggregation of
control or data. Such transformations typically
10   break up abstractions created by the programmer,
or construct new bogus abstractions by bundling
together unrelated data or control.

Similarly, some transformations affect the
ordering of data or control. In many cases the
15   order in which two items are declared or two
computations are performed has no effect on the
observable behavior of the program. There can,
however, be much useful information embedded in
the chosen order, to the programmer who wrote the
20   program as well as to a reverse engineer. The
closer two items or events are in space or time,
the higher the likelihood that they are related in
one way or another. Ordering transformations try
to explore this by randomizing the order of
25   declarations or computations.

5 Evaluating Obfuscating Transformations
Before we can attempt to design any
obfuscating transformations, we should be able to
30   evaluate the quality of such a transformation. In

-29-

SUBSTITUTE SHEET (RULE 26)

this section we will attempt to classify
transformations according to several criteria: how
much obscurity they add to the program (e.g.,
potency), how difficult they are to break for a
5   deobfuscator (e.g., resilience), and how much
computational overhead they add to the obfuscated
application (e.g., cost).

5.1   Measures of Potency
10   We will first define what it means for a
program P' to be more obscure (or complex or
unreadable) than a program P. Any such metric
will, by definition, be relatively vague, because
it must be based (in part) on human cognitive
15 abilities.

Fortunately, we can draw upon the vast body
of work in the Software Complexity Metrics branch
of Software Engineering. In this field, metrics
are designed with the intent to aid the
20 construction of readable, reliable, and
maintainable software. The metrics are frequently
based on counting various textual properties of
the source code and combining these counts into a
measure of complexity. While some of the formulas
25 that have been proposed have been derived from
empirical studies of real programs, others have
been purely speculative.

The detailed complexity formulas found in the
metrics' literature can be used to derive general
30 statements, such as: "if programs P and P' are

-30-

identical except that P' contains more of property
q than P, then P' is more complex than P." Given
such a statement, we can attempt to construct a
transformation that adds more of the q-property to
5   a program, knowing that this is likely to increase
its obscurity.

FIG. 7 is a table that tabulates some of the
more popular complexity measures, in which E(x) is
the complexity of a software component x, F is a
10   function or method, C is a class, and P is a
program. When used in a software construction
project the typical goal is to minimize these
measures. In contrast, when obfuscating a program
we generally want to maximize the measures.

15   The complexity metrics allow us to formalize
the concept of potency and will be used below as a
measure of the usefulness of a transformation.
Informally, a transformation is potent if it does
a good job confusing Bob, by hiding the intent of
20   Alice's original code. In other words, the potency
of a transformation measures how much more
difficult the obfuscated code is to understand
(for a human) than the original code. This is
formalized in the following definition:

25

Definition 2 (Transformation Potency) Let T be a
behavior-conserving transformation, such that
P $-^{T}->$ P' transforms a source program P into a
target program P'. Let E(P) be the complexity of
30   P, as defined by one of the metrics of FIG. 7.

-31-

$T_{pot}(P)$, the potency of T with respect to a program P, is a measure of the extent to which T changes the complexity of P . It is defined as

$$T_{pot}(P) \overset{def}{=} E(P')/E(P) - 1.$$

5   T is a potent obfuscating transformation if $T_{pot}(P) > 0$.

For the purposes of this discussion, we will measure potency on a three-point scale, (low, 10   medium, high).

The observations in Table 1 make it possible for us to list some desirable properties of a transformation T. In order for T to be a potent obfuscating transformation, it should

15   - increase overall program size (u_1) and introduce new classes and methods (u^a_7).
            - introduce new predicates (u_2) and increase the nesting level of conditional and looping constructs
20   (u_3).
            - increase the number of method arguments (u_5) and inter-class instance variable dependencies (u^d_7).
            - increase the height of the inheritance
25   tree ($u^{b,c}_7$).
            - increase long-range variable dependencies (u_4).

-32-

SUBSTITUTE SHEET (RULE 26)

5.2 Measures of Resilience

At first glance it would seem that increasing $T_{pot}(P)$ would be trivial. To increase the u_2 metric, for example, all we have to do is to add
5   some arbitrary if-statements to P:

```
main() {              main() {
S1;                   S1;
S2;        =ᵀ=>       if (5==2) S1;
                          S2;}
                      if (1>2) S2;

                      }
```

Unfortunately, such transformations are
15   virtually useless, because they can easily be undone by simple automatic techniques. It is therefore necessary to introduce the concept of resilience, which measures how well a transformation holds up under attack from an
20   automatic deobfuscator. For example, the resilience of a transformation T can be seen as the combination of two measures:

Programmer Effort: the amount of time
25   required to construct an automatic deobfuscator that is able to effectively reduce the potency of T , and

Deobfuscator Effort: the execution time and
30   space required by such an automatic

-33-

deobfuscator to effectively reduce the
potency of T .

It is important to distinguish between
5   resilience and potency. A transformation is potent
if it manages to confuse a human reader, but it is
resilient if it confuses an automatic
deobfuscator.

We measure resilience on a scale from trivial
10   to one way, as shown in FIG. 8a. One-way
transformations are special, in the sense that
they can never be undone. This is typically
because they remove information from the program
that was useful to the human programmer, but which
15   is not necessary in order to execute the program
correctly. Examples include transformations that
remove formatting, and scramble variable names.

Other transformations typically add useless
information to the program that does not change
20   its observable behavior, but which increases the
"information load" on a human reader. These
transformations can be undone with varying degrees
of difficulty.

FIG. 8b shows that deobfuscator effort is
25   classified as either polynomial time or
exponential time. Programmer effort, the work
required to automate the deobfuscation of a
transformation T, is measured as a function of the
scope of T. This is based on the intuition that
30   it is easier to construct counter-measures against

-34-

an obfuscating transformation that only affects a
small part of a procedure, than against one that
may affect an entire program.

The scope of a transformation is defined
5   using terminology borrowed from code optimization
theory: T is a local transformation if it affects
a single basic block of a control flow graph
(CFG), it is global if it affects an entire CFG,
it is inter-procedural if it affects the flow of
10   information between procedures, and it is an
interprocess transformation if it affects the
interaction between independently executing
threads of control.

15   Definition 3 (Transformation Resilience) Let T be
a behavior-conserving transformation, such that
$P =^T=> P'$ transforms a source program P into a
target program P'. $T_{res}(P)$ is the resilience of T
with respect to a program P.
20          $T_{res}(P)$ is a one-way transformation if
information is removed from P such that P cannot
be reconstructed from P'. Otherwise,
      $T_{res} \stackrel{def}{=} Resilience(T_{Deobfuscator\ effort}, T_{Programmer\ effort}),$
25   in which Resilience is the function defined in the
matrix in FIG. 8b.

5.3 Measures of Execution Cost
      In FIG. 2b, we see that potency and
30   resilience are two of the three components

-35-

describing the quality of a transformation. The
third component, the cost of a transformation, is
the execution time or space penalty that a
transformation incurs on an obfuscated

5   application. We classify the cost on a four-point
scale (free, cheap, costly, dear), in which each
point is defined below:

Definition 5 (Transformation Cost) Let T be a

10   behavior-conserving transformation, such that
$T_{cost}(P) \in$ {dear, costly, cheap, free} with $T_{cost}(P)$
= free, if executing P' requires $O(1)$ more
resources than P; otherwise $T_{cost}(P)$ = cheap, if
executing P' requires $O(n)$ more resources than P;

15   otherwise $T_{cost}(P)$ = costly, if executing P'
requires $O(n^p)$, with p>1, more resources than P;
otherwise $T_{cost}(P)$ = dear (i.e., executing P'
requires exponentially more resources than P).

It should be noted that the actual cost

20   associated with a transformation depends on the
environment in which it is applied. For example, a
simple assignment statement a=5 inserted at the
top-most level of a program will only incur a
constant overhead. The same statement inserted

25   inside an inner loop will have a substantially
higher cost. Unless noted otherwise, we always
provide the cost of a transformation as if it had
been applied at the outermost nesting level of the
source program.

SUBSTITUTE SHEET (RULE 26)

5.4   Measures of Quality

We can now give a formal definition of the quality of an obfuscating transformation:

5

Definition 6 (Transformation Quality)
$T_{qual}(P)$, the quality of a transformation T, is defined as the combination of the potency, resilience, and cost of T: $T_{qual}(P) = (T_{pot}(P),$

10   $T_{res}(P), T_{cost}(P))$.

5.5   Layout Transformations

Before we explore novel transformations, we will briefly consider the trivial layout

15   transformations, which, for example, are typical of current Java obfuscators such as Crema. (Hans Peter Van Vliet. Crema --- The Java obfuscator. http://web.inter.nl.net/users/H.P.van. Vliet/crema.html, January 1996) The first

20   transformation removes the source code formatting information sometimes available in Java class files. This is a one-way transformation, because once the original formatting is gone it cannot be recovered; it is a transformation with low

25   potency, because there is very little semantic content in formatting, and no great confusion is introduced when that information is removed; finally, this is a free transformation, because the space and time complexity of the application

30   is not affected.

-37-

Scrambling identifier names is also a one-way
and free transformation. However, it has a much
higher potency than formatting removal, because
identifiers contain a great deal of pragmatic
5   information.

6 Control Transformations

In this and the next few sections we will
present a catalogue of obfuscating
10   transformations. Some have been derived from
well-known transformations used in other
areas such as compiler optimization and software
reengineering, others have been developed for the
sole purpose of obfuscation, in accordance with
15   one embodiment of the present invention.

In this section we will discuss
transformations that attempt to obscure the
control-flow of the source application. As
indicated in FIG. 2f, we classify these
20   transformations as affecting the aggregation,
ordering, or computations of the flow of control.
Control aggregation transformations break up
computations that logically belong together or
merge computations that do not. Control ordering
25   transformations randomize the order in which
computations are carried out. Computation
transformations can insert new (redundant or dead)
code, or make algorithmic changes to the source
application.

-38-

SUBSTITUTE SHEET (RULE 26)

For transformations that alter the flow of
control, a certain amount of computational
overhead will be unavoidable. For Alice this
means that she may have to choose between a highly
5   efficient program, and one that is highly
obfuscated. An obfuscator can assist her in this
trade-off by allowing her to choose between cheap
and expensive transformations.

10   6.1 Opaque Predicates
The real challenge when designing
control-altering transformations is to make them
not only cheap, but also resistant to attack from
deobfuscators. To achieve this, many
15   transformations rely on the existence of opaque
variables and opaque predicates. Informally, a
variable V is opaque if it has some property q
that is known a priori to the obfuscator, but
which is difficult for the deobfuscator to deduce.
20   Similarly, a predicate P (a Boolean expression) is
opaque if a deobfuscator can deduce its outcome
only with great difficulty, while this outcome is
well known to the obfuscator.
Being able to create opaque variables and
25   predicates that are difficult for a deobfuscator
to crack is a major challenge to a creator of
obfuscation tools, and the key to highly resilient
control transformations. We measure the resilience
of an opaque variable or predicate (i.e., its
30   resistance to deobfuscation attacks) on the same

-39-

SUBSTITUTE SHEET (RULE 26)

scale as transformation resilience (i.e., trivial,
weak, strong, full, one-way). Similarly, we
measure the added cost of an opaque construct on
the same scale as transformation cost (i.e.,

5   free, cheap, costly, dear).

Definition 7 (Opaque Constructs) A variable V is
opaque at a point p in a program, if V has a
property q at p, which is known at obfuscation

10   time. We write this as V^q_p or V^q if p is clear from
the context. A predicate P is opaque at p if its
outcome is known at obfuscation time. We write P^F_p
(P^T_p) if P always evaluates to False (True) at p,
and $P^?_p$ if P sometimes evaluates to True and

15   sometimes to False. Again, p will be omitted if
clear from the context. FIG. 9 shows different
types of opaque predicates. Solid lines indicate
paths that may sometimes be taken, and dashed
lines indicate paths that will never be taken.

20   Below we give some examples of simple opaque
constructs. These are easy to construct for the
obfuscator and equally easy to crack for the
deobfuscator. Section 8 provides examples of
opaque constructs with much higher resilience.

25

6.1.1 Trivial and Weak Opaque Constructs
        An opaque construct is trivial if a
deobfuscator can crack it (i.e.,deduce its value)
by a static local analysis. An analysis is local

30   if it is restricted to a single basic block of a

-40-

control flow graph. FIGs. 10a and 10b provide
examples of (a) trivial opaque constructs and (b)
weak opaque constructs.

We also consider an opaque variable to be
5   trivial if it is computed from calls to library
functions with simple, well-understood semantics.
For a language like the Java™, language which
requires all implementations to support a standard
set of library classes, such opaque variables are
10   easy to construct. A simple example is int $v^s[1,5]$
= random(1,5), in which random(a, b) is a library
function that returns an integer in the range a .
. . b. Unfortunately, such opaque variables are
equally easy to deobfuscate. All that is required
15   is for the deobfuscator-designer to tabulate the
semantics of all simple library functions, and
then pattern-match on the function calls in the
obfuscated code.

An opaque construct is weak if a deobfuscator
20   can crack it by a static global analysis. An
analysis is global if it is restricted to a single
control flow graph.

6.2 Computation Transformations
25   Computation Transformations fall into three
categories: hide the real control-flow behind
irrelevant statements that do not contribute to
the actual computations, introduce code sequences
at the object code level for which there exist no
30   corresponding high-level language constructs, or

-41-

remove real control-flow abstractions or introduce
spurious ones.

6.2.1 Insert Dead or Irrelevant Code

5   The u_2 and u_3 metrics suggest that there is a
strong correlation between the perceived
complexity of a piece of code and the number of
predicates it contains. Using opaque predicates,
we can devise transformations that introduce new
10   predicates in a program.

Consider the basic block $S = S_1$ . . . Sn in
FIG. 11. In FIG. 11a, we insert an opaque
predicate P^T into S, essentially splitting it in
half. The P^T predicate is irrelevant code, because
15   it will always evaluate to True. In FIG. 11b, we
again break S into two halves, and then proceed to
create two different obfuscated versions S^a and S^b
of the second half. S^a and S^b will be created by
applying different sets of obfuscating
20   transformations to the second half of S. Hence, it
will not be directly obvious to a reverse engineer
that S^a and S^b in fact perform the same function.
We use a predicate $P^?$ to select between S^a and S^b
at runtime.

25   FIG. 11c is similar to FIG. 11b, but this
time we introduce a bug into S^b. The P^T predicate
always selects the correct version of the code, S^a.

6.2.2 Extend Loop Conditions

-42-

SUBSTITUTE SHEET (RULE 26)

FIG. 12 shows how we can obfuscate a loop by making the termination condition more complex. The basic idea is to extend the loop condition with a p^T or p^F predicate that will not affect the number

5   of times the loop will execute. The predicate we have added in FIG. 12d, for example, will always evaluate to True because $x^2(x+1)^2=0\,(\mathrm{mod}4)$.

6.2.3 Convert a Reducible to a Non-Reducible Flow

10   Graph

Often, a programming language is compiled to a native or virtual machine code, which is more expressive than the language itself. When this is the case, it allows us to devise language-breaking

15   transformations. A transformation is language-breaking if it introduces virtual machine (or native code) instruction sequences that have no direct correspondence with any source language construct. When faced with such instruction

20   sequences a deobfuscator will either have to try to synthesize an equivalent (but convoluted) source language program or give up altogether.

For example, the Java™ bytecode has a goto instruction, but the Java™ language has no

25   corresponding goto statement. This means that the Java™ bytecode can express arbitrary control flow, whereas the Java™ language can only (easily) express structured control flow. Technically, we say that the control flow graphs produced from

30   Java™ programs will always be reducible, but the

-43-

Java™ bytecode can express non-reducible flow
graphs.

Since expressing non-reducible flow graphs
becomes very awkward in languages without gotos,
5   we construct a transformation that converts a
reducible flow graph to a non-reducible one. This
can be done by turning a structured loop into a
loop with multiple headers. For example, in FIG.
13a, we add an opaque predicate P^F to a while
10   loop, to make it appear that there is a jump into
the middle of the loop. In fact, this branch will
never be taken.

A Java™ decompiler would have to turn a
non-reducible flow graph into one which either
15   duplicates code or which contains extraneous
Boolean variables. Alternatively, a deobfuscator
could guess that all non-reducible flow graphs
have been produced by an obfuscator, and simply
remove the opaque predicate. To counter this we
20   can sometimes use the alternative transformation
shown in FIG 13b. If a deobfuscator blindly
removes P^F, the resulting code will be incorrect.

In particular, FIGs. 13a and 13b illustrate a
transformation for transforming a Reducible flow
25   graph to a Non-Reducible Flow graph. In FIG. 13a,
we split the loop body S2 into two parts (S^a_2 and
S^b_2), and insert a bogus jump to the beginning of
S^b_2. In FIG. 13b, we also break S1 into two parts,
S^a_1 and S^b_1. S^b_1 is moved into the loop and an opaque
30   predicate P^T ensures that S^b_1 is always executed

-44-

SUBSTITUTE SHEET (RULE 26)

before the loop body. A second predicate Q^F ensures that S^b. is only executed once.

6.2.4 Remove Library Calls and Programming Idioms

5   Most programs written in Java rely heavily on calls to the standard libraries. Because the semantics of the library functions are well known, such calls can provide useful clues to a reverse engineer. The problem is exacerbated by the fact
10 that references to Java library classes are always by name, and these names cannot be obfuscated.

In many cases the obfuscator will be able to counter this by simply providing its own versions of the standard libraries. For example, calls to
15 the Java Dictionary class (which uses a hash table implementation) could be turned into calls to a class with identical behavior, but implemented as, for example, a red-black tree. The cost of this transformation is not so much in execution time,
20 but in the size of the program.

A similar problem occurs with clichés (or patterns), common programming idioms that occur frequently in many applications. An experienced reverse engineer will search for such patterns to
25 jump-start his understanding of an unfamiliar program. As an example, consider linked lists in Java™. The Java™ library has no standard class that provides common list operations such as insert, delete, and enumerate. Instead, most Java™
30 programmers will construct lists of objects in an

-45-

ad hoc fashion by linking them together on a next
field.   Iterating through such lists is a very
common pattern in Java™ programs. Techniques
invented in the field of automatic program

5   recognition (see Linda Mary Wills. Automated
program recognition: a feasibility demonstration.
Artificial Intelligence, 45(1--2):113--172, 1990,
incorporated herein by reference) can be used to
identify common patterns and replace them with

10   less obvious ones. In the linked list case, for
example, we might represent the standard list data
structure with a less common one, such as cursors
into an array of elements.

15   6.2.5 Table Interpretation
        One of the most effective (and expensive)
transformations is table interpretation.   The idea
is to convert a section of code (Java bytecode in
this example) into a different virtual machine

20   code. This new code is then executed by a virtual
machine interpreter included with the obfuscated
application. Obviously, a particular application
can contain several interpreters, each accepting a
different language and executing a different

25   section of the obfuscated application.
        Because there is usually an order of
magnitude slow down for each level of
interpretation, this transformation should be
reserved for sections of code that make up a small

-46-

part of the total runtime or which need a very
high level of protection.

6.2.6 Add Redundant Operands

5   Once we have constructed some opaque
variables we can use algebraic laws to add
redundant operands to arithmetic expressions. This
will increase the u_1 metric. Obviously, this
technique works best with integer expressions
10   where numerical accuracy is not an issue. In the
obfuscated statement (1') below we make use of an
opaque variable P whose value is 1. In statement
(2') we construct an opaque subexpression P/Q
whose value is 2. Obviously, we can let P and Q
15   take on different values during the execution of
the program, as long as their quotient is 2
whenever statement (2') is reached.

(1) X=X+V; $=^T=>$ (1') X=X+V*P$^{=1}$;

20   (2) Z=L+1;   (2') Z=L+(P$^{=2Q}$/Q$^{=P/2}$)/2.

6.2.7 Parallelize Code

        Automatic parallelization is an important
compiler optimization used to increase the
25   performance of applications running on
multi-processor machines. Our reasons for wanting
to parallelize a program, of course, are
different. We want to increase parallelism not to
increase performance, but to obscure the actual

-47-

flow of control. There are two possible operations
available to us:

       1.   We can create dummy processes that
5   perform no useful task, and

       2.   We can split a sequential section of
           the application code into multiple
           sections executing in parallel.

10

    If the application is running on a
single-processor machine, we can expect these
transformations to have a significant execution
time penalty. This may be acceptable in many
15 situations, because the resilience of these
transformations is high: static analysis of
parallel programs is very difficult, because the
number of possible execution paths through a
program grows exponentially with the number of
20 executing processes. Parallelization also yields
high levels of potency: a reverse engineer will
find a parallel program much more difficult to
understand than a sequential one.

    As shown in FIG. 14, a section of code can be
25 easily parallelized if it contains no data
dependencies. For example, if S_1 and S_2 are two
data-independent statements they can be run in
parallel. In a programming language like the
Java™ language that has no explicit parallel

SUBSTITUTE SHEET (RULE 26)

constructs, programs can be parallelized using
calls to thread (lightweight process) libraries.

    As shown in FIG. 15, a section of code that
contains data dependencies can be split into
5   concurrent threads by inserting appropriate
synchronization primitives, such as await and
advance (see Michael Wolfe. High Performance
Compilers For Parallel Computing. Addison-Wesley,
1996. ISBN 0-8053-2730-4, incorporated herein by
10   reference). Such a program will essentially be
running sequentially, but the flow of control will
be shifting from one thread to the next.

6.3 Aggregation Transformations

15   Programmers overcome the inherent complexity
of programming by introducing abstractions. There
is abstraction on many levels of a program, but
the procedural abstraction is the most important
one. For this reason, obscuring procedure and
20   method calls can be important to the obfuscator.
Below, we will consider several ways in which
methods and method invocations can be obscured:
inlining, outlining, interleaving, and cloning.
The basic idea behind all of these is the same:
25   (1) code that the programmer aggregated into a
method (presumably because it logically belonged
together) should be broken up and scattered over
the program and (2) code that seems not to belong
together should be aggregated into one method.

-49-

6.3.1 Inline and Outline Methods

Inlining is, of course, a important compiler
optimization. It is also an extremely useful

5   obfuscation transformation, because it removes
procedural abstractions from the program. Inlining
is a highly resilient transformation (it is
essentially one-way), because once a procedure
call has been replaced with the body of the called

10   procedure and the procedure itself has been
removed, there is no trace of the abstraction left
in the code. FIG. 16 shows how procedures P and Q
are inlined at their call-sites, and then removed
from the code.

15   Outlining (turning a sequence of statements
into a subroutine) is a very useful companion
transformation to inlining. We create a bogus
procedural abstraction by extracting the beginning
of Q's code and the end of P's code into a new

20   procedure R.

In object-oriented languages such as the
Java™ language, inlining may, in fact, not always
be a fully one-way transformation. Consider a
method invocation m.P(). The actual procedure

25   called will depend on the run-time type of m. In
cases when more than one method can be invoked at
a particular call site, we inline all possible
methods (see Jeffrey Dean. Whole-Program
Optimization of Object-Oriented Languages. PhD

30   thesis, University of Washington, 1996,

-50-

incorporated herein by reference) and select the
appropriate code by branching on the type of m).
Hence, even after inlining and removal of methods,
the obfuscated code may still contain some traces

5   of the original abstractions. For example, FIG.
17 illustrates inlining method calls. Unless we
can statically determine the type of m, all
possible methods to which m.P() could be bound
must be inlined at the call site.

10

6.3.2 Interleave Methods

    The detection of interleaved code is an
important and difficult reverse engineering task.

    FIG. 18 shows how we can interleave two

15 methods declared in the same class. The idea is to
merge the bodies and parameter lists of the
methods and add an extra parameter (or global
variable) to discriminate between calls to the
individual methods. Ideally, the methods should be

20 similar in nature to allow merging of common code
and parameters. This is the case in FIG. 18, in
which the first parameter of M1 and M2 have
the same type.

25 6.3.3 Clone Methods

    When trying to understand the purpose of a
subroutine a reverse engineer will of course
examine its signature and body. However, equally
important to understanding the behavior of the

30 routine are the different environments in which it

SUBSTITUTE SHEET (RULE 26)

is being called. We can make this process more
difficult by obfuscating a method's call sites to
make it appear that different routines are being
called, when, in fact, this is not the case.

5   FIG. 19 shows how we can create several
different versions of a method by applying
different sets of obfuscating transformations to
the original code. We use method dispatch to
select between the different versions at runtime.

10   Method cloning is similar to the predicate
insertion transformations in FIG. 11, except that
here we are using method dispatch rather than
opaque predicates to select between different
versions of the code.

15

6.3.4 Loop Transformations
        A large number of loop transformations have
been designed with the intent to improve the
performance of (in particular) numerical
20 applications. See Bacon [2] for a comprehensive
survey. Some of these transformations are useful
to us, because they also increase the complexity
metrics, which are discussed above with respect to
FIG. 7. Loop Blocking, as shown in FIG. 20a, is
25 used to improve the cache behavior of a loop by
breaking up the iteration space so that the inner
loop fits in the cache. Loop unrolling, as shown
in FIG. 20b, replicates the body of a loop one or
more times. If the loop bounds are known at
30 compile time the loop can be unrolled in its

-52-

SUBSTITUTE SHEET (RULE 26)

entirety. Loop fission, as shown in FIG. 20c,
turns a loop with a compound body into several
loops with the same iteration space.

All three transformations increase the u_1 and
5   u_2 metrics, because they increase the source
application's total code size and number of
conditions. The loop blocking transformation also
introduces extra nesting, and hence also increases
the u_3 metric.

10   Applied in isolation, the resilience of these
transformations is quite low. It does not require
much static analysis for a deobfuscator to reroll
an unrolled loop. However, when the
transformations are combined, the resilience rises
15   dramatically. For example, given the simple loop
in FIG. 20b, we could first apply unrolling, then
fission, and finally blocking. Returning the
resulting loop to its original form would require
a fair amount of analysis for the deobfuscator.

20

6.4 Ordering Transformations

Programmers tend to organize their source
code to maximize its locality. The idea is that a
program is easier to read and understand if two
25   items that are logically related are also
physically close in the source text. This kind of
locality works on every level of the source: for
example, there is locality among terms within
expressions, statements within basic blocks, basic
30   blocks within methods, methods within classes, and

-53-

classes within files. All kinds of spatial
locality can provide useful clues to a reverse
engineer. Therefore, whenever possible, we
randomize the placement of any item in the source

5   application. For some types of items (methods
within classes, for example) this is trivial. In
other cases (such as statements within basic
blocks) a data dependency analysis (see David F.
Bacon, Susan L. Graham, and Oliver J. Sharp.

10   Compiler transformations for high-performance
computing. ACM Computing Surveys, 26(4):345--420,
December 1994. http://
www.acm.org/pubs/toc/Abstracts/0360-0300/
197406.html. and Michael Wolfe. High Performance

15   Compilers For Parallel Computing. Addison-Wesley,
1996. ISBN 0-8053-2730-4, incorporated herein by
reference) is performed to determine which
reorderings are technically valid.

      These transformations have low potency (they

20   do not add much obscurity to the program), but
their resilience is high, in many cases one-way.
For example, when the placement of statements
within a basic block has been randomized, there
will be no traces of the original order left in

25   the resulting code.

      Ordering transformations can be particularly
useful companions to the "Inline-Outline"
transformation of Section 6.3.1. The potency of
that transformation can be enhanced by (1)

30   inlining several procedure calls in a procedure P,

-54-

SUBSTITUTE SHEET (RULE 26)

(2) randomizing the order of the statements in P,
and (3) outlining contiguous sections of P's
statements. This way, unrelated statements that
were previously part of several different

5   procedures are brought together into bogus
procedural abstractions.

      In certain cases it is also possible to
reorder loops, for example by running them
backwards. Such loop reversal transformations are

10   common in high-performance compilers (David F.
.Bacon, Susan L. Graham, and Oliver J. Sharp.
Compiler transformations for high-performance
computing. ACM Computing Surveys, 26(4):345--420,
December 1994. http://

15   www.acm.org/pubs/toc/Abstracts/0360-0300/
197406.html.).

7 Data Transformations
      In this section we will discuss

20   transformations that obscure the data structures
used in the source application. As indicated in
FIG. 2e, we classify these transformations as
affecting the storage, encoding, aggregation, or
ordering of the data.

25

7.1 Storage and Encoding Transformations
      In many cases there is a "natural" way to
store a particular data item in a program. For
example, to iterate through the elements of an

30   array we probably would choose to allocate a local

integer variable of the appropriate size as the
iteration variable. Other variable types might be
possible, but they would be less natural and
probably less efficient.

5   Furthermore, there is also often a "natural"
interpretation of the bit-patterns that a
particular variable can hold which is based on the
type of the variable. For example, we would
normally assume that a 16-bit integer variable

10   storing the bit-pattern 0000000000001100 would
represent the integer value 12. Of course, these
are mere conventions and other interpretations are
possible.

Obfuscating storage transformations attempt

15   to choose unnatural storage classes for dynamic as
well as static data. Similarly, encoding
transformations attempt to choose unnatural
encodings for common data types. Storage and
encoding transformations often go hand-in-hand,

20   but they can sometimes be used in isolation.

7.1.1 Change Encoding

As a simple example of an encoding
transformation we will replace an integer variable

25   i by $i_0 = c_1 * i + c_2$, where c_1 and c_2 are
constants. For efficiency, we could choose c_1 to be
a power of two. In the example below, we let $c_1 = 8$
and $c_2 = 3$:

-56-

```
    {                   =ᵀ=>          {
      int i=1;                          int i=i1;
      while (i < 1000)                  while (i < 8003)
          . . . A[i] . . .;                 . . A[(i-3)/8] . . .;
5         i++;                              i+=8;
    }                                   }
```

Obviously, overflow (and, in case of floating point variables, accuracy) issues need to be

10 addressed. We could either determine that because of the range of the variable (the range can be determined using static analysis techniques or by querying the user) in question no overflow will occur, or we could change to a larger variable

15 type.

There will be a trade-off between resilience and potency on one hand, and cost on the other. A simple encoding function such as $i_0 = c_1 + i + c_2$ in the example above, will add little extra

20 execution time but can be deobfuscated using common compiler analysis techniques (Michael Wolfe. High Performance Compilers For Parallel Computing. Addison-Wesley, 1996. ISBN 0-8053-2730-4. and David F. Bacon, Susan L.

25 Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. ACM Computing Surveys, 26(4):345--420, December 1994. http:// www.acm.org/pubs/toc/Abstracts/0360-0300/

30 197406.html).

-57-

7.1.2 Promote Variables

There are a number of simple storage
transformations that promote variables from a
5   specialized storage class to a more general class.
Their potency and resilience are generally low,
but used in conjunction with other transformations
they can be quite effective. For example, in Java,
an integer variable can be promoted to an integer
10   object. The same is true of the other scalar types
which all have corresponding "packaged" classes.
Because Java™ supports garbage collection, the
objects will be automatically removed when they
are no longer referenced. Here is an example:

15

```
{                           {
int I=1;                    int i = new int(1);
while (i < 9)    =⁴=> while (i.value < 9)
      ...A[i]...;                 ...A[i.value]...;
20      i++;                       i.value++;
}                           }
```

It is also possible to change the lifetime of
a variable. The simplest such transform turns a
25   local variable into a global one which is then
shared between independent procedure invocations.
For example, if procedures P and Q both reference
a local integer variable, and P and Q cannot both
be active at the same time (unless the program
30   contains threads, this can be determining by

-58-

examining the static call graph) then the variable
can be made global and shared between them:

```
     void P() {          int C;
 5   int i; ...I...      void P(){
     }                   ...C...
                         }
     void Q(){    =ᵀ=>   while (i.value<9)
     int k;...k...       ...C...
10   }                   }
```

This transformation increases the u_s metric,
because the number of global data structures
referenced by P and Q is increased.

15

7.1.3 Split Variables

Boolean variables and other variables of
restricted range can be split into two or more
variables. We will write a variable V split into k
20 variables p_1, . . . , p_k as V = [p_1, . . . , p_k].
Typically, the potency of this transformation will
grow with k. Unfortunately, so will the cost of
the transformation, so we usually restrict k to 2
or 3.

25   To allow a variable V of type T to be split
into two variables p and q of type U requires us
to provide three pieces of information: (1) a
function f(p; q) that maps the values of p and q
into the corresponding value of V, (2) a function
30 g(V) that maps the value of V into the

-59-

corresponding values of p and q, and (3) new
operations (corresponding to the primitive
operations on values of type T) cast in terms of
operations on p and q. In the remainder of this

5   section we will assume that V is of type Boolean,
and p and q are small integer variables.

FIG. 21a shows a possible choice of
representation for split Boolean variables. The
table indicates that if V has been split into p

10   and q, and if, at some point in the program, p = q
= 0 or p = q = 1, then that corresponds to V being
False. Similarly, p = 0, q = 1 or p = 1, q = 0
corresponds to True.

Given this new representation, we have to

15   devise substitutions for various built-in Boolean
operations (e.g., &, or). One approach is to
provide a run-time lookup table for each operator.
Tables for "AND" and "OR" are shown in FIGs. 21c
and 21d, respectively. Given two Boolean variables

20      $V_1 = [p, q]$ and $V_2 = [r, s]$, V_1 & V_2 is computed as
AND[2p + q, 2r + s].

In FIG. 21e, we show the result of splitting
three Boolean variables A=[a1,a2], B=[b1,b2], and
C=[c1,c2]. An interesting aspect of our chosen

25   representation is that there are several possible
ways to compute the same Boolean expression.
Statements (3') and (4') in FIG. 21e, for example,
look different, although they both assign False to
a variable. Similarly, while statements (5') and

-60-

(6') are completely different, they both compute A
& B.

The potency, resilience, and cost of this
transformation all grow with the number of
5   variables into which the original variable is
split. The resilience can be further enhanced by
selecting the encoding at run-time. In other
words, the run-time look-up tables of FIGs. 21b
through 21d are not constructed at compile-time
10   (which would make them susceptible to static
analyses) but by algorithms included in the
obfuscated application. This, of course, would
prevent us from using in-line code to compute
primitive operations, as done in statement (6') in
15   FIG. 21e.

7.1.4 Convert Static to Procedural Data

Static data, particularly character strings,
contain much useful pragmatic information to a
20   reverse engineer. A technique for obfuscating a
static string is to convert it into a program that
produces the string. The program -- which could be
a DFA or a Trie traversal -- could possibly
produce other strings as well.
25   As an example, consider a function G of FIG.
22, which is constructed to obfuscate the strings
''AAA'', ''BAAAA'', and ''CCB''. The values
produced by G are G(1)=''AAA'', G(2)=''BAAAA'',
G(3)=G(5)=''CCB'', and G(4)=''XCB'' (which is not

-61-

actually used in the program). For other argument
values, G may or may not terminate.

Aggregating the computation of all static
string data into just one function is, of course,

5   highly undesirable. Much higher potency and
resilience is achieved if the G-function was
broken up into smaller components that were
embedded into the ``normal'' control flow of the
source program.

10   It is interesting to note that we can combine
this technique with the table interpretation
transformation of Section 6.2.5. The intent of
that obfuscation is to convert a section of Java
bytecode into code for another virtual machine.

15   The new code will typically be stored as static
string data in the obfuscated program. For even
higher levels of potency and resilience, however,
the strings could be converted to programs that
produce them, as discussed above.

20

7.2 Aggregation Transformations

In contrast to imperative and functional
languages, object-oriented languages are more
data-oriented than control-oriented. In other

25   words, in an object-oriented program, the control
is organized around the data structures, rather
than the other way around. This means that an
important part of reverse-engineering an
object-oriented application is trying to restore

30   the program's data structures. Conversely, it is

-62-

important for an obfuscator to try to hide these
data structures.

In most object-oriented languages, there are
just two ways to aggregate data: in arrays and in
5   objects. In the next three sections we will
examine ways in which these data structures can be
obfuscated.

7.2.1 Merge Scalar Variables

10   Two or more scalar variables V_1 . . . V_k can
be merged into one variable V_M , provided the
combined ranges of V_1 . . . V_k will fit within the
precision of V_M. For example, two 32-bit integer
variables could be merged into one 64-bit
15   variable. Arithmetic on the individual variables
would be transformed into arithmetic on V_M. As a
simple example, consider merging two 32-bit
integer variables X and Y into a 64-bit variable
Z. Using the merging formula,

20

$$Z(X, Y) = 2^{32} * Y + X$$

we get the arithmetic identities in FIG. 23a. Some
simple examples are given in FIG. 23b.
25   In particular, FIG. 23 shows merging two
32-bit variables X and Y into one 64-bit variable
Z. Y occupies the top 32 bits of Z, X the bottom
32 bits. If the actual range of either X or Y can
be deduced from the program, less intuitive merges
30   could be used. FIG. 23a gives rules for addition

-63-

and multiplication with X and Y. FIG. 23b shows
some simple examples. The example could be
further obfuscated, for example by merging (2')
and (3') into Z+=47244640261.

5   The resilience of variable merging is quite
low. A deobfuscator only needs to examine the set
of arithmetic operations being applied to a
particular variable in order to guess that it
actually consists of two merged variables. We can

10   increase the resilience by introducing bogus
operations that could not correspond to any
reasonable operations on the individual variables.
In the example in FIG. 23b, we could insert
operations that appear to merge Z's two halves,

15   for example, by rotation: if (P^F) Z = rotate(Z,5).

A variant of this transformation is to merge
V_1 . . . V_k into an array

$$V_A \quad = \qquad 1 \; . \; . \; . \; k$$

20                                      $$V_1 \; . \; . \; . \; V_k$$

of the appropriate type. If V_1 . . . V_k are object
reference variables, for example, then the element
type of VA can be any class that is higher in the

25   inheritance hierarchy than any of the types of
V_1 . . . V_k.

7.2.2 Restructure Arrays

A number of transformations can be devised

30   for obscuring operations performed on arrays: for

-64-

example, we can split an array into several
sub-arrays, merge two or more arrays into one
array, fold an array (increasing the number of
dimensions), or flatten an array (decreasing the
5   number of dimensions).

    FIG. 24 shows some examples of array
restructuring. In statements (1-2) an array A is
split up into two sub-arrays A1 and A2. A1 holds
the elements of A that have even indices, and A2
10   holds the elements with odd indices.

    Statements (3-4) of FIG. 24 show how two
integer arrays B and C can be interleaved into a
resulting array BC. The elements from B and C are
evenly spread over the resulting array.

15   Statements (6-7) demonstrate how a
one-dimensional array D can be folded into a two-
dimensional array D1. Statements (8-9), finally,
demonstrate the reverse transformation: a two-
dimensional array E is flattened into a
20   one-dimensional array E1.

    Array splitting and folding increase the u_6
data complexity metric. Array merging and
flattening, on the other hand, seem to decrease
this measure. While this may seem to indicate that
25   these transformations have only marginal or even
negative potency, this, in fact, is deceptive. The
problem is that the complexity metrics of FIG. 7
fail to capture an important aspect of some data
structure transformations: they introduce
30   structure where there was originally none or they

-65-

remove structure from the original program. This
can greatly increase the obscurity of the program.
For example, a programmer who declares a
two-dimensional array does so for a purpose: the
5   chosen structure somehow maps cleanly to the data
that is being manipulated. If that array is folded
into a one-dimensional structure, a reverse
engineer will have been deprived of much valuable
pragmatic information.

10

7.2.3 Modify Inheritance Relations
        In current object-oriented language such as
the Java™ language, the main modularization and
abstraction concept is the class. Classes are
15   essentially abstract data types that encapsulate
data (instance variables) and control (methods).
We write a class as $C = (V, M)$, where V is the set
of C's instance variables and M its methods.
        In contrast to the traditional notion of
20   abstract data types, two classes C_1 and C_2 can be
composed by aggregation (C_2 has an instance
variable of type C_1) as well as by inheritance (C_2
extends C_1 by adding new methods and instance
variables). We write inheritance as $C_2 = C_1 \cup C'_2$.
25   C_2 is said to inherit from C_1, its super- or parent
class. The U operator is the function that
combines the parent class with the new properties
defined in C'_2. The exact semantics of U depends on
the particular programming language. In languages
30   such as Java, U is usually interpreted as union

-66-

when applied to the instance variables and as
overriding when applied to methods.

According to metric u_7, the complexity of a
class C_1 grows with its depth (distance from the
5   root) in the inheritance hierarchy and the number
of its direct descendants. For example, there are
two basic ways in which we can increase this
complexity: we can split up (factor) a class as
shown in FIG. 25a or insert a new, bogus, class as
10   shown in FIG. 25b.

A problem with class factoring is its low
resilience; there is nothing stopping a
deobfuscator from simply merging the factored
classes. To prevent this, factoring and insertion
15   are normally combined as shown in FIG. 25d.
Another way of increasing the resilience of these
types of transformations is to make sure that new
objects are created of all introduced classes.

FIG. 25c shows a variant of class insertion,
20   called false refactoring. Refactoring is a
(sometimes automatic) technique for restructuring
object-oriented programs whose structure has
deteriorated (see William F. Opdyke and Ralph E.
Johnson. Creating abstract superclasses by
25   refactoring. In Stan C. Kwasny and John F. Buck,
editors, Proceedings of the 21st Annual Conference
on Computer Science, pages 66--73, New York, NY,
USA, February 1993. ACM Press.
ftp://st.cs.uiuc.edu/pub/papers/refactoring/refact
30   oring-superclasses.ps, incorporated herein by

-67-

reference). Refactoring is a two-step process.
First, it is detected that two, apparently
independent classes, in fact implement similar
behavior. Secondly, features common to both

5   classes are moved into a new (possibly abstract)
parent class. False refactoring is a similar
operation, only it is performed on two classes C_1
and C_2 that have no common behavior. If both
classes have instance variables of the

10   same type, these can be moved into the new parent
class C_3. C_3's methods can be buggy versions of
some of the methods from C_1 and C_2.

      7.3 Ordering Transformations

15   In Section 6.4 we showed that (when possible)
randomizing the order in which computations are
performed is a useful obfuscation. Similarly, it
is useful to randomize the order of declarations
in the source application.

20   Particularly, we randomize the order of
methods and instance variables within classes and
formal parameters within methods. In the latter
case, the corresponding actuals will of course
have to be reordered as well. The potency of these

25   transformations is low and the resilience is
one-way.

           In many cases it will also be possible to
reorder the elements within an array. Simply put,
we provide an opaque encoding function f(i) which

maps the i:th element in the original array into
its new position of the reordered array:

```
     {                              {
 5   int i=1, A[1000];              int i=1, A[1000];
     while (i < 1000)    =ᵀ=>       while (i < 1000)
          ...A[i]...;                    ...A[f(i)]...;
          i++;                           i++;
     }                              }
```

10

8 Opaque Values and Predicates

As we have seen, opaque predicates are the
major building block in the design of
transformations that obfuscate control flow. In
15 fact, the quality of most control transformations
is directly dependent on the quality of such
predicates.

In Section 6.1 we gave examples of simple
opaque predicates with trivial and weak
20 resilience. This means that the opaque predicates
can be broken (an automatic deobfuscator could
determine their value) using local or global
static analysis. Obviously, we generally require a
much higher resistance to attack. Ideally, we
25 would like to be able to construct opaque
predicates that require worst case exponential
time (in the size of the program) to break but
only polynomial time to construct. In this section
we will present two such techniques. The first one

SUBSTITUTE SHEET (RULE 26)

is based on aliasing and the second is based on
lightweight processes.

8.1 Opaque Constructs Using Objects and Aliases

5   Inter-procedural static analysis is
significantly complicated whenever there is a
possibility of aliasing. In fact, precise,
flow-sensitive alias analysis is undecidable in
languages with dynamic allocation, loops, and
10   if-statements.

         In this section we will exploit the
difficulty of alias analysis to construct opaque
predicates that are cheap and resilient to
automatic deobfuscation attacks.

15

8.2 Opaque Constructs Using Threads

         Parallel programs are more difficult to
analyze statically than their sequential
counterparts. The reason is their interleaving
20   semantics: n statements in a parallel region PAR
S_1, S_2. . . ., S_n, ENDPAR can be executed in n!
different ways. In spite of this, some
static analyses over parallel programs can be
performed in polynomial time [18], while others
25   require all n! interleavings to be considered.

         In Java, parallel regions are constructed
using lightweight processes known as threads. Java
threads have (from our point of view) two very
useful properties: (1) their scheduling policy is
30   not specified strictly by the language

-70-

specification and will hence depend on the
implementation, and (2) the actual scheduling of a
thread will depend on asynchronous events, such as
generated by user interaction, and network

5   traffic. Combined with the inherent interleaving
semantics of parallel regions, this means that
threads are very difficult to analyze statically.

    We will use these observations to create
opaque predicates (see FIG. 32) that will require

10   worst-case exponential time to break. The basic
idea is very similar to the one used in Section
8.2: a global data structure V is created and
occasionally updated, but kept in a state such
that opaque queries can be made. The difference is

15   that V is updated by concurrently executing
threads.

    Obviously, V can be a dynamic data structure
such as the one created in FIG. 26. The threads
would randomly move the global pointers g and h

20   around in their respective components, by
asynchronously executing calls to move and insert.
This has the advantage of combining data races
with interleaving and aliasing effects, for very
high degrees of resilience.

25   In FIG. 27, we illustrate these ideas with a
much simpler example where V is a pair of global
integer variables X and Y. It is based on the
well-known fact from elementary number theory
that, for any integers x and y, $7y^2 -1$ does not

30   equal x^2.

-71-

9 Deobfuscation and Preventive Transformations

Many of our obfuscating transformations (particularly the control transformations of

5   Section 6.2) can be said to embed a bogus program within a real program. In other words, an obfuscated application really consists of two programs merged into one: a real program which performs a useful task and a bogus program which

10   computes useless information. The sole purpose of the bogus program is to confuse potential reverse engineers by hiding the real program behind irrelevant code.

The opaque predicate is the main device the

15   obfuscator has at its disposal to prevent the bogus inner program from being easily identified and removed. For example, in FIG. 28a, an obfuscator embeds bogus code protected by opaque predicates within three statements of a real

20   program. A deobfuscator's task is to examine the obfuscated application and automatically identify and remove the inner bogus program. To accomplish this, the deobfuscator must first identify and then evaluate opaque constructs. This process is

25   illustrated in FIGs.28b through 28d.

FIG. 29 shows the anatomy of a semi-automatic deobfuscation tool. It incorporates a number of techniques that are well known in the reverse engineering community. In the remainder of this

30   section we will briefly review some of these

-72-

techniques and discuss various counter-measures
(so called preventive transformations) that an
obfuscator can employ to make deobfuscation more
difficult.

5

9.1 Preventive Transformations

Preventive transformations, which are
discussed above with respect to FIG. 2g, are quite
different in flavor from control or data
10   transformations. In contrast to these, their main
goal is not to obscure the program to a human
reader. Rather, they are designed to make known
automatic deobfuscation techniques more difficult
(inherent preventive transformations), or to
15   explore known problems in current deobfuscators or
decompilers (targeted preventive transformations).

9.1.1 Inherent Preventive Transformations

Inherent preventive transformations will
20   generally have low potency and high resilience.
Most importantly, they will have the ability to
boost the resilience of other transformations. As
an example, assume that we have reordered a
for-loop to run backwards, as suggested in section
25   6.4. We were able to apply this transformation
only because we could determine that the loop had
no loop-carried data dependencies. Naturally,
there is nothing stopping a deobfuscator from
performing the same analysis and then returning
30   the loop to forward execution. To prevent this, we

-73-

can add a bogus data dependency to the reversed
loop:

```
{                                 {
5   for(i=1;i<=10;i++)   =T=>        int B[50];
        A[i]=i                       for(i=10;i<=1;i--)
    }                                A[i]=i;
                                     B[i]+=B[i*i/2]

                                 }
10
```

The resilience this inherent preventive
transformation adds to the loop reordering
transformation depends on the complexity of the
bogus dependency and the state-of-the-art in
15 dependency analysis [36].

9.1.2 Targeted Preventive Transformations

As an example of a targeted preventive
transformation, consider the HoseMocha program
20 (Mark D. LaDue. HoseMocha. http://www.xynyx.
demon.nl/java/HoseMocha.java, January 1997). It
was designed specifically to explore a weakness in
the Mocha (Hans Peter Van Vliet. Mocha --- The
Java decompiler. http://web.inter.nl.net/users/H.
25 P.van.Vliet/mocha.html, January 1996) decompiler.
HoseMocha inserts extra instructions after every
return-statement in every method in the source
program. This transformation has no effect on the
behavior of the application, but it is enough to
30 make Mocha crash.

-74-

9.2 Identifying and Evaluating Opaque Constructs

The most difficult part of deobfuscation is identifying and evaluating opaque constructs. Note that identification and evaluation are distinct activities. An opaque construct can be local (contained within a single basic block), global (contained within a single procedure), or inter-procedural (distributed throughout the entire program). For example, if $(x * x == (7^F * y * y -1))$ is a local opaque predicate, whereas $R=X*X;$ . . .;$S=7*y*y-1;$. . .;if $(R == S^F)$. . . is global. If the computation of R and S were performed in different procedures, the construct would be inter-procedural. Obviously, identification of a local opaque predicate is easier than identification of an inter-procedural one.

9.3 Identification by Pattern Matching

A deobfuscator can use knowledge of the strategies employed by known obfuscators to identify opaque predicates. A designer of a deobfuscator could examine an obfuscator (either by decompiling it or simply by examining the obfuscated code it generates) and construct pattern-matching rules that can identify commonly used opaque predicates. This method will work best for simple local predicates, such as $x * x == (7 * y * y -1)$ or $random(1^F,5) < 0$

-75-

To thwart attempts at pattern matching, the
obfuscator should avoid using canned opaque
constructs. It is also important to choose opaque
constructs that are syntactically similar to the
5   constructs used in the real application.

9.4 Identification by Program Slicing
     A programmer will generally find the
obfuscated version of a program more difficult to
10   understand and reverse engineer than the original
one. The main reasons are that in the obfuscated
program (a) live "real" code will be interspersed
with dead bogus code, and (b) logically related
pieces of code will have been broken up and
15   dispersed over the program. Program slicing tools
can be used by a reverse engineer to counter these
obfuscations. Such tools can interactively aid the
engineer to decompose a program into manageable
chunks called slices. A slice of a program P with
20   respect to a point p and a variable v consists of
all the statements of P that could have
contributed to v's value at p. Hence, a program
slicer would be able to extract from the
obfuscated program the statements of the algorithm
25   that computes an opaque variable v, even if the
obfuscator has dispersed these statements over the
entire program.
     There are several strategies available to an
obfuscator to make slicing a less useful
30   identification tool: Add parameter aliases A

-76-

SUBSTITUTE SHEET (RULE 26)

parameter alias is two formal parameters (or a
formal parameter and a global variable) that refer
to the same memory location. The cost of precise
inter-procedural slicing grows with the number of

5   potential aliases in a program, which in turn
grows exponentially with the number of formal
parameters. Hence, if the obfuscator adds aliased
dummy parameters to a program it will either
substantially slow down the slicer (if precise

10   slices are required), or force the slicer to
produce imprecise slices (if fast slicing is
required).

      Add variable dependencies, as popular slicing
tools such as Unravel (James R. Lyle, Dolorres R.

15   Wallace, James R. Graham, Keith B. Gallagher,
Joseph P. Poole, and David W Binkley. Unravel: A
CASE tool to assist evaluation of high integrity
software. Volume 1: Requirements and design.
Technical Report NIS-

20   TIR 5691, U.S. Department of Commerce, August
1995) work well for small slices, but will
sometimes require excessive time to compute larger
ones. For example, when working on a 4000 line C
program, Unravel in some cases required over 30

25   minutes to compute a slice. To force this
behavior, the obfuscator should attempt to
increase slice sizes, by adding bogus variable
dependencies. In the example below, we have
increased the size of the slice computing

x by adding two statements which apparently
contribute to x's value, but which, in fact, do
not.

```
5   main()      {              =ᵀ=>        main() {
    int x=1;                            int x=1;
    x = x * 3;                              if (Pᶠ ) x++;
    }                                   x = x + V0 ;
                                        x = x * 3;
10                                      }
```

9.5 Statistical Analysis

A deobfuscator can instrument an obfuscated
program to analyze the outcome of all predicates.
15 We will call any deobfuscation method that
examines the run-time characteristics of an
obfuscated application in this way, Statistical
Analysis. The deobfuscator would alert the reverse
engineer to any predicate that always returns the
20 same truth value over a large number of test runs,
because they may turn out to be an opaque P^T (P^F)
predicate. The deobfuscator could not blindly
replace such predicates with True (False), because
this would be too dangerous. Many applications
25 will contain ``real'' predicates that check for
conditions that only happen under exceptional
circumstances, and to the deobfuscator they will
appear to behave identically to an opaque
predicate. As an example, consider pif (Leap Year)
30 . . . .

-78-

Statistical analysis can also be used for
evaluation. When a potential opaque predicate
(e.g., P^T) in a program M has been identified, we
guess its value (True), and make a version M' of
5   the obfuscated program where the opaque predicate
has been replaced by the guessed value. We then
run M and M' in parallel on the same input, and
compare to see that they produce identical output.
If the outputs are the same, we can conclude that
10 the predicate was part of the bogus, not the real,
application, as shown in FIG. 30.

We have to make sure that our chosen inputs
adequately cover all paths in the program. Again,
if the program contains paths that are rarely
15 taken (if (Leap Year) . . .) this will be
difficult. Furthermore, generating large numbers
of correct input/output data is very difficult,
particularly when internal structure of the
application is unknown, or the input is entered
20 (as is often the case with Java programs) through
a complex graphical user interface.

To prevent identification by statistical
analysis, the obfuscator may choose to favor
transformations that insert $P^?$ predicates (such as
25 the one shown in FIG. 11b) over those that insert
P^T or P^F predicates.

Another possible counter-measure against
statistical analysis is to design opaque
predicates in such a way that several predicates
30 have to be cracked at the same time. One way of

-79-

doing this is to let the opaque predicates have
side-effects. In the example below the obfuscator
has determined (through some sort of static flow
analysis) that statements S_1 and S_2 must always
5   execute the same number of times. The statements
are obfuscated by introducing opaque predicates
which are calls to functions Q_1 and Q_1. Q_1 and Q_2
increment and decrement a global variable k:

```
10   {                =ᵀ=>        {

     S₁;                         int k=0;

     S₂;                         bool Q₁ (x) {

     }                           k+=2³¹ ; return (Pᵀ₁ )}

                                 bool Q₂ (x) {

15                               k-=2³¹ ; return (P⁻₂ )}

                                 {

                                 if (Q₁ (j) T ) S₁ ;

                                          . . .

                                 if (Q₂ (k) T ) S₂ ;

20                               }
```

        If the deobfuscator tries to replace one (but
not both) predicates with True, k will overflow.
As a result, the deobfuscated program will
25   terminate with an error condition.

9.6 Evaluation by Data-Flow Analysis
        Deobfuscation is similar to many types of
code optimization. Removing if (False) . . . is
30   dead code elimination and moving identical code

SUBSTITUTE SHEET (RULE 26)

from if-statement branches (e.g., S_1 and S_0^1 in
FIG. 28) is code hoisting, both common code
optimization techniques.

When an opaque construct has been identified
5   we can attempt to evaluate it. In simple cases
constant propagation using a reaching definition
data-flow analysis can be sufficient: x=5;. .
.;y=7;. . .;if (x*x==(7*y*y-1)) . . . .

10 9.7 Evaluation by Theorem Proving

If data-flow analysis is not powerful enough
to break the opaque predicate, a deobfuscator can
attempt to use a theorem prover. Whether this is
doable or not depends on the power of
15 state-of-the-art theorem provers (which is
difficult to ascertain) and the complexity of the
theorem that needs to be proven. Certainly,
theorems that can be proved by induction (such as
$x^2(x + 1)^2 = 0$ (mod 4)), are well within reach of
20 current theorem provers.

To make things more difficult, we can use
theorems which are know to be difficult to prove,
or for which no known proof exists. In the example
below the deobfuscator will have to prove that the
25 bogus loop always terminates in order to determine
that S_2 is live code:

{ $=^T=>$ {
S_1; S_1;

SUBSTITUTE SHEET (RULE 26)

```
    S₂;                              n = random(1,2³²);
    }                                do
                                         n = ((n%2)!=0)?3*n+1:n/2
                                     while (n>1);
5                                    S₂;
                                     }
```

This is known as the Collatz problem. A conjecture
says that the loop will always terminate. Although
10   there is no known proof of this conjecture, the
code is known to terminate for all numbers up to
$7*10^{11}$. Thus, this obfuscation is safe (the
original and obfuscated code behave identically),
but is difficult to deobfuscate.

15

9.8 Deobfuscation and Partial Evaluation
        Deobfuscation also resembles partial
evaluation. A partial evaluator splits a program
into two parts: the static part which can be
20   precomputed by the partial evaluator, and the
dynamic part which is executed at runtime. The
dynamic part would correspond to our original,
unobfuscated, program. The static part would
correspond to our bogus inner program, which, if
25   it were identified, could be evaluated and removed
at deobfuscation time.
        Like all other static inter-procedural
analysis methods, partial evaluation is sensitive
to aliasing. Hence, the same preventive

transformations that were discussed in relation to
slicing also applies to partial evaluation.

10 Obfuscation Algorithms

5   Given the obfuscator architecture of Section
3, the definition of obfuscation quality in
Section 5, and the discussion of various
obfuscating transformations in Section 6 through
Section 9, we are now in a position to present
10 more detailed algorithms, in accordance with one
embod8iment of the present invention.
        The top-level loop of an obfuscation tool can
have this general structure:

15 WHILE NOT Done(A) DO
        S := SelectCode(A);
        T := SelectTransform(S);
        A := Apply(T ,S);
    END;
20

    SelectCode returns the next source code object to
be obfuscated. SelectTransform returns the
transformation which should be used to obfuscate
the particular source code object. Apply applies
25 the transformation to the source code object and
updates the application accordingly. Done
determines when the required level of obfuscation
has been attained. The complexity of these
functions will depend on the sophistication of the
30 obfuscation tool. At the simplistic end of the

-83-

scale, SelectCode and SelectTransform could simply
return random source code object/transformations,
and Done could terminate the loop when the size of
the applica-
5   tion exceeds a certain limit. Normally, such
behavior is insufficient.

Algorithm 1 gives a description of a code
obfuscation tool with a much more sophisticated
selection and termination behavior. In one
10   embodiment, the algorithm makes use of several
data structures, which are constructed by
Algorithms 5, 6, and 7:

P_s For each source code object S, $P_s(S)$

15   is the set of language constructs the
programmer used in S. $P_s(S)$ is used to
find appropriate obfuscating
transformations for S.

20   A For each source code object S, $A(S) =$
$\{T_1 \dashrightarrow V_1 ; \ldots ; T_n \dashrightarrow V_n\}$ is a
mapping from transformations T_i to values
V_i, describing how appropriate
it would be to apply T_i to S. The idea is
25   that certain transformations may be
inappropriate for a particular source
code object S, because they introduce
new code which is ``unnatural'' to S.
The new code would look out of place in
30   S and hence would be easy to spot for a

-84-

reverse engineer. The higher the

appropriateness value V_i the better the

code introduced by transformation T_i will

fit in.

5

I For each source code object S, I(S) is

the obfuscation priority of S. I(S)

describes how important it is to

obfuscate the contents of S. If S

10   contains an important trade secret then

I(S) will be high, if it contains mainly

``bread-and-butter'' code I(S) will be

low.

15   R For each routine M , R(M) is the

execution time rank of M . R(M) = 1 if

more time is spent executing M than any

other routine.

20   The primary input to Algorithm 1 is an

application A and a set of obfuscating

transformations { T_1 ; T_2 ; . . .}. The algorithm

also requires information regarding each

transformation, particularly three quality

25   functions $T_{res}(S)$, $T_{pot}(S)$, and $T_{cost}(S)$ (similar to

their namesakes in Section 5, but returning

numerical values) and a function P_t :

$T_{res}(S)$ returns a measure of the resilience of

30   transformation T when applied to source code

-85-

object S (i.e., how well T will withstand an
attack from an automatic deobfuscator).

T_{pot} (S) returns a measure of the potency of
5   transformation T when applied to source code
object S (i.e., how much more difficult S
will be for a human
to understand after having been obfuscated by
T).

10

T_{cost} (S) returns a measure of the execution
time and space penalty added by T to S.

P_t maps each transformation T to the set of
15   language constructs that T will add to the
application.

Points 1 to 3 of Algorithm 1 load the application
to be obfuscated, and builds appropriate internal
20   data structures. Point 4 builds P_s(S), A(S), I(S),
and R(M). Point 5 applies obfuscating
transformations until the required obfuscation
level has been attained or until the maximum
execution time penalty is exceeded. Point 6,
25   finally, rewrites the new application A'.

Algorithm 1 (Code Obfuscation)

input: a) An application A made up of source

SUBSTITUTE SHEET (RULE 26)

code or object code files C1 ; C2 ; . .
. .

b) The standard libraries L1 ; L2 ; . .
. de-

fined by the language.

c) A set of obfuscating transformations
{T1 ; T2 ; . . .}.

d) A mapping Pt which, for each
transformation T gives the set of
language constructs that T will add to
the application.

e) Three functions $T_{res}(S)$, $T_{pot}(S)$,
$T_{cost}(S)$ expressing the quality of a
transformation T with respect to a
source code object S.

f) A set of input data $I = \{I1; I2; . .
. \}$ to A.

g) Two numeric values AcceptCost>0 and
ReqObf>0. AcceptCost is a measure of the
maximum extra execution time/space
penalty the user will accept. ReqObf is
a measure of the amount of obfuscation
required by the user.

output: An obfuscated application A' made up
of source code or object code files.

1. Load the application C_1 ; C_2 ; . . . to be
obfuscated. The obfuscator could either

(a) load source code files, in which case the obfuscator would have to contain a complete compiler front-end performing lexical,

5   syntactic, and semantic analysis, (a less powerful obfuscator that restricts itself to purely syntactic transformation could manage without semantic analysis) or

10   (b) load object code files. If the object code retains most or all of the information in the source code (as is the case with Java class files), this method is preferable.

15   2. Load library code files L1; L2; . . . referenced directly or indirectly by the application.

3. Build an internal representation of the
20   application. The choice of internal representation depends on the structure of the source language and the complexity of the transformations the obfuscator implements. A typical set of data structures might include:

25   (a) A control-flow graph for each routine in A.

       (b) A call-graph for the routines in A.

       (c) An inheritance graph for the classes in A.

-88-

SUBSTITUTE SHEET (RULE 26)

4. Construct mappings R(M) and P_s(S) (using

Algorithm 5), I(S) (using Algorithm 6), and A(S)

(using Algorithm 7).

5

5. Apply the obfuscating transformations to the

application. At each step we select a source code

object S to be obfuscated and a suitable

transformation T to apply to S. The

10   process terminates when the required obfuscation

level has been reached or the accept-

able execution time cost has been exceeded.

REPEAT

15   S := SelectCode(I);

       T := SelectTransform(S, A);

       Apply T to S and update relevant data

       structures from point 3;

   UNTIL Done(ReqObf, AcceptCost, S, T , I).

20

6. Reconstitute the obfuscated source code objects

into a new obfuscated application, A'.

Algorithm 2 (SelectCode)

25

input: The obfuscation priority mapping I as

               computed by Algorithm 6.

output: A source code object S.

I maps each source code object S to I(S),
which is a measure of how important it is to
obfuscate S. To select the next source code object
5   to obfuscate, we can treat I as a priority queue.
In other words, we select S so that I(S) is
maximized.

Algorithm 3 (SelectTransform)
10
input: a) A source code object S.

b) The appropriateness mapping A as
computed by Algorithm 7.

15   output: A transformation T .

Any number of heuristics can be used to
select the most suitable transformation to apply
to a particular source code object S. However,
20   there are two important issues to consider.
Firstly, the chosen transformation must blend in
naturally with the rest of the code in S. This can
be handled by favoring transformations with a high
appropriateness value in A(S). Secondly, we want
25   to favor transformations which yield a high
'bang-for-the-buck' (i.e. high levels of
obfuscation with low execution time penalty). This
is accomplished by selecting transformations that
maximize potency and resilience, and minimize
30   cost. These heuristics are captured by the

-90-

following code, where w1, w2, w3 are

implementation-defined constants:

       Return a transform T , such that

5   T --> V is within A(S), and

       $(w1*T_{pot}(S) + w2*T_{res}(S) +w3*V) / T_{cost}(S)$

       is maximized.

Algorithm 4 (Done)

10

input:

       a) ReqObf, the remaining level of

       obfuscation.

       b) AcceptCost, the remaining acceptable

15   execution time penalty.

       c) A source code object S.

       d) A transformation T .

       e) The obfuscation priority mapping I .

20 output:

       a) An updated ReqObf.

       b) An updated AcceptCost.

       c) An updated obfuscation priority mapping I.

       d) A Boolean return value which is TRUE if

25   the termination condition has been

       reached.

       The Done function serves two purposes. It

updates the priority queue I to reflect the fact

30 that the source code object S has been obfuscated,

and should receive a reduced priority value. The
reduction is based on a combination of the
resilience and potency of the transformation. Done
also updates ReqObf and AcceptCost, and determines

5   whether the termination condition has been
reached. w_1, w_2, w_3, w_4 are implementation-defined
constants:

$$I(S) := I(S) - (w_2 T_{pos}(S) + w_2 T_{res}(S));$$

10 $$ReqObf := ReqObf - (w_2 T_{pos}(S) + w_2 T_{res}(S));$$

$$AcceptCost := AcceptCost - T_{cost}(s);$$

$$RETURN\ AcceptCost <= 0\ OR\ ReqObf <= 0.$$

Algorithm 5 (Pragmatic Information)

15

input:

    a) An application A.

    b) A set of input data I = {I1 ; I2 ; . . .}

    to A.

20

output:

    a) A mapping R(M) which, for every routine M

    in A, gives the execution time rank of M .

    b). A mapping P s (S), which, for every

25   source code object S in A, gives the set

    of language constructs used in S.

Compute pragmatic information.   This
information will be used to choose the right type

of transformation for each particular source code object.

    1. Compute dynamic pragmatic information (i.e., run the application under a profiler on the input data set I provided by the user. Compute $R(M)$ (the execution time rank of M) for each routine/basic block, indicating where the application spends most of its time.

    2. Compute static pragmatic information $P_s(S)$. $P_s(S)$ provides statistics on the kinds of language constructs the programmer used in S.

```
FOR S := each source code object in A DO
        O := The set of operators that S uses;
        C := The set of high-level language
        constructs (WHILE statements,
        exceptions, threads, etc.) that S uses;
        L := The set of library classes/routines
        that S references;
        Ps(S) := O U C U L;
END FOR.
```

Algorithm 6 (Obfuscation Priority)

input:

    a) An application A.

b) R(M), the rank of M.

output: A mapping I(S) which, for each source
            code object S in A, gives the obfuscation
5   priority of S.

I(S) can be provided explicitly by the user,
or it can be computed using a heuristic based on
the statistical data gathered in Algorithm 5.
10   Possible heuristics might be:

1. For any routine M in A, let I(M) be
   inversely proportional to the rank of M , R(M
). I.e. the idea is that ``if much time is
15   spent executing a routine M , then M is
   probably an important procedure that should
   be heavily obfuscated.''

2. Let I(S) be the complexity of S, as
20   defined by one of the software complexity
   metrics in Table 1. Again, the (possibly
   flawed) intuition is that complex code is
   more likely to contain important trade
   secrets than simple code.
25

Algorithm 7 (Obfuscation Appropriateness)

input:
   a) An application A.

-94-

b) A mapping P t which, for each
transformation T , gives the set of language
constructs T will add to the application.

c) A mapping P s (S) which, for each source

5   code object S in A, gives the set of language
constructs used in S.

output: A mapping A(S) which, for each source code
object S in A and each transformation T ,

10   gives the appropriateness of T with respect
to S.

Compute the appropriateness set A(S) for each
source code object S. The mapping is based

15   primarily on the static pragmatic information
computed in Algorithm 5.


```
FOR S := each source code object in A DO
        FOR T := each transformation DO
20              V := degree of similarity between
                Pt(T) and Ps(S);
                A(S) := A(S) U {T --> V};
        END FOR
END FOR
```

25

11 Summary and Discussion

We have observed that it may under many
circumstances be acceptable for an obfuscated
program to behave differently than the original

30   one. In particular, most of our obfuscating

-95-

transformations make the target program slower or
larger than the original. In special cases we even
allow the target program to have different
side-effects than the original, or not to

5   terminate when the original program terminates
with an error condition. Our only requirement is
that the observable behavior (the behavior as
experienced by a user) of the two programs should
be identical.

10   Allowing such weak equivalence between
original and obfuscated program is a novel and
very exciting idea. Although various
transformations are provided and described above,
many other transformations will be apparent to one

15   of ordinary skill in the art and can be used to
provide obfuscation for enhanced software security
in accordance with the present invention.

There is also great potential for much future
research to identify transformations not yet

20   known. In particular, we would like to see the
following areas investigated:

1. New obfuscating transformations should be
identified.

25

2. The interaction and ordering between
different transformations should be studied.
This is similar to work in code optimization,
where the ordering of a sequence of

-96-

SUBSTITUTE SHEET (RULE 26)

optimizing transformations has always been a
difficult problem.

3. The relationship between potency and cost
should be studied. For a particular kind of
code we would like to know which
transformations would give the best
"bang-for-the-buck" (i.e., the highest
potency at the lowest execution overhead).

For an overview of all the transformations
that have been discussed above, see FIG. 31. For
an overview of the opaque constructs that have
been discussed above, see FIG. 32. However, the
present invention should not be limited to the
exemplary transformations and opaque constructs
discussed above.

11.1 The Power of Obfuscation
Encryption and program obfuscation bear a
striking resemblance to each other. Not only do
both try to hide information from prying eyes,
they also purport to do so for a limited time
only. An encrypted document has a limited
shelf-life: it is safe only for as long as the
encryption algorithm itself withstands attack, and
for as long as advances in hardware speed do not
allow messages for the chosen key-length to be
routinely decrypted. The same is true for an
obfuscated application; it remains secret only for

-97-

as long as sufficiently powerful deobfuscators
have yet to be built.

For evolving applications this will not be a
problem, as long as the time between releases is
5   shorter than the time it takes for the
deobfuscator to catch up with the obfuscator. If
this is the case, then by the time an application
can be automatically deobfuscated it is already
outdated and of no interest to a competitor.

10   However, if an application contains trade
secrets that can be assumed to survive several
releases, then these should be protected by means
other than obfuscation. Partial server-side
execution (Figure 2(b)) seems the obvious choice,
15   but has the drawback that the application will
execute slowly or (when the network connection is
down) not at all.

11.2 Other Uses of Obfuscation
20   It is interesting to note that there may be
potential applications of obfuscation other than
as discussed above. One possibility is to use
obfuscation in order to trace software pirates.
For example, a vendor creates a new obfuscated
25   version of his application for every new customer
(We can generate different obfuscated versions of
the same application by introducing an element of
randomness into the SelectTransform algorithm
(Algorithm 3). Different seeds to the random
30   number generator will produce different versions.)

-98-

and keeps a record of to whom each version was
sold. This is probably only reasonable if the
application is being sold and distributed over the
net. If the vendor finds out that his application

5   is being pirated, all he needs to do is to get a
copy of the pirated version, compare it against
the data base, and see who bought the original
application. It is, in fact, not necessary to
store a copy of every obfuscated version sold. It

10 suffices to keep the random number seed that was
sold.

Software pirates could themselves make
(illicit) use of obfuscation. Because the Java
obfuscator we outlined above works at the bytecode

15 level, there is nothing stopping a pirate from
obfuscating a legally bought Java application. The
obfuscated version could then be resold. When
faced with litigation the pirate could argue that
he is, in fact, not reselling the application that

20 he originally bought (after all, the code is
completely different!), but rather a legally
reengineered version.

25   Conclusion

In conclusion, the present invention provides a
computer implemented method and apparatus for
preventing, or at least hampering, reverse engineering
of software. While this may be effected at the expense

30 of execution time or program size with the resulting

transformed program behaving differently at a detailed
level, it is believed that the present technique
provides significant utility in appropriate
circumstances. In one embodiment, the transformed
5   program has the same observable behavior as the
untransformed program. Accordingly, the present
invention allows for such weak equivalence between the
original and obfuscated program.

While the present discussion has been primarily in
10   the context of hampering reverse engineering of
software, other applications are contemplated such as
watermarking software objects (including applications).
This exploits the potentially distinctive nature of any
single obfuscation procedure. A vendor would create a
15   different obfuscated version of an application for
every customer sold. If pirate copies are found, the
vendor need only compare it against the original
obfuscation information database to be able to trace
the original application.

20   The particular obfuscation transformations
described herein are not exhaustive. Further
obfuscation regimes may be identified and used in the
present novel obfuscation tool architecture.

Where in the foregoing description reference has
25   been made to elements or integers having known
equivalents, then such equivalents are included as if
they were individually set forth.

Although the present invention has been described
by way of example and with reference to particular
30   embodiments. It is to be understood that modifications

SUBSTITUTE SHEET (RULE 26)

and improvements can be made without departing from the
scope of the present invention.

SUBSTITUTE SHEET (RULE 26)

CLAIMS

What is claimed is:

5   1.   A computer implemented method for obfuscating
code, comprising:
            selecting a subset of the code to obfuscate;
            selecting an obfuscating transform to apply;
and
10   applying the transformation, wherein the
transformed code provides weak equivalence to the
untransformed code.

2.   The computer implemented method of Claim 1,
15   further comprising:
            identifying one or more source code input
files corresponding to source code for the code of
an application to be processed;
            selecting a required level of obfuscation
20   (the potency);
            selecting a maximum execution time or space
penalty (the cost);
            reading and parsing the input files;
            providing information identifying data types,
25   data structures, and control structures used by
the application to be processed;
            selecting and applying obfuscating
transformations to source code objects until the
required potency has been achieved or the maximum
30   cost has been exceeded; and

-102-

SUBSTITUTE SHEET (RULE 26)

outputting the transformed code of the
application.

3.   The method of Claim 1, wherein the
5   transformation comprises an opaque construct, the
opaque construct being constructed using aliasing and
concurrency techniques.

4.   The method of Claim 1, further comprising:
10   outputting information about obfuscating
transformations applied to the obfuscated code and
information relating obfuscated code of a
transformed application to source code of the
application.

15

5.   The method of Claim 1, wherein the
transformation is selected to preserve the observable
behavior of the code of an application.

20   6.   The method of Claim 1, further comprising:
deobfuscating the code, the deobfuscating the
code comprising removing any obfuscations from the
obfuscated code of an application by use of
slicing, partial evaluation, dataflow analysis, or
25   statistical analysis.

7.   A computer program embodied on a computer-
readable medium for obfuscating code, comprising:
logic that selects a subset of the code to
30   obfuscate;

logic that selects an obfuscating transform
to apply; and

logic that applies the transformation,
wherein the transformed code provides weak

5   equivalence to the untransformed code.

8.   The computer program of Claim 7, further
comprising:

logic that identifies one or more source code

10   input files corresponding to source code for the
code of an application to be processed;

logic that selects a required level of
obfuscation (the potency);

logic that selects a maximum execution time

15   or space penalty (the cost);

logic that reads and parses the input files;

logic that provides information identifying
data types, data structures, and control
structures used by the application to be

20   processed;

logic that selects and applies obfuscating
transformations to source code objects until the
required potency has been achieved or the maximum
cost has been exceeded; and

25   logic that outputs the transformed code of
the application.

9.   The computer program of Claim 7, wherein the
transformation comprises an opaque construct, the

opaque construct being constructed using aliasing and
concurrency techniques.

10.   The computer program of Claim 7, further
comprising:
        logic that outputs information about
        obfuscating transformations applied to the
        obfuscated code and information relating
        obfuscated code of a transformed application to
        source code of the application.

11.   The computer program of Claim 7, wherein the
transformation is selected to preserve the observable
behavior of the code of an application.

12.   The computer program of Claim 7, further
comprising:
        logic that deobfuscates the code, the
        deobfuscating the code comprising removing any
        obfuscations from the obfuscated code of an
        application by use of slicing, partial evaluation,
        dataflow analysis, or statistical analysis.

13.   An apparatus for obfuscating code,
comprising:
        means for selecting a subset of the code to
        obfuscate;
        means for selecting an obfuscating transform
        to apply; and

SUBSTITUTE SHEET (RULE 26)

means for applying the transformation,
wherein the transformed code provides weak
equivalence to the untransformed code.

5   14. The apparatus of Claim 13, further
comprising:
        means for identifying one or more source code
input files corresponding to source code for the
code of an application to be processed;
10   means for selecting a required level of
obfuscation (the potency);
        means for selecting a maximum execution time
or space penalty (the cost);
        means for reading and parsing the input
15   files;
        means for providing information identifying
data types, data structures, and control
structures used by the application to be
processed;
20   means for selecting and applying obfuscating
transformations to source code objects until the
required potency has been achieved or the maximum
cost has been exceeded; and
        . means for outputting the transformed code of
25   the application.

        15. The apparatus of Claim 13, wherein the
transformation comprises an opaque construct, the
opaque construct being constructed using aliasing and
30   concurrency techniques.

SUBSTITUTE SHEET (RULE 26)

16.   The apparatus of Claim 13, further
comprising:

            means for outputting information about
5   obfuscating transformations applied to the
         obfuscated code and information relating
         obfuscated code of a transformed application to
         source code of the application.

10   17.   The apparatus of Claim 13, wherein the
transformation is selected to preserve the observable
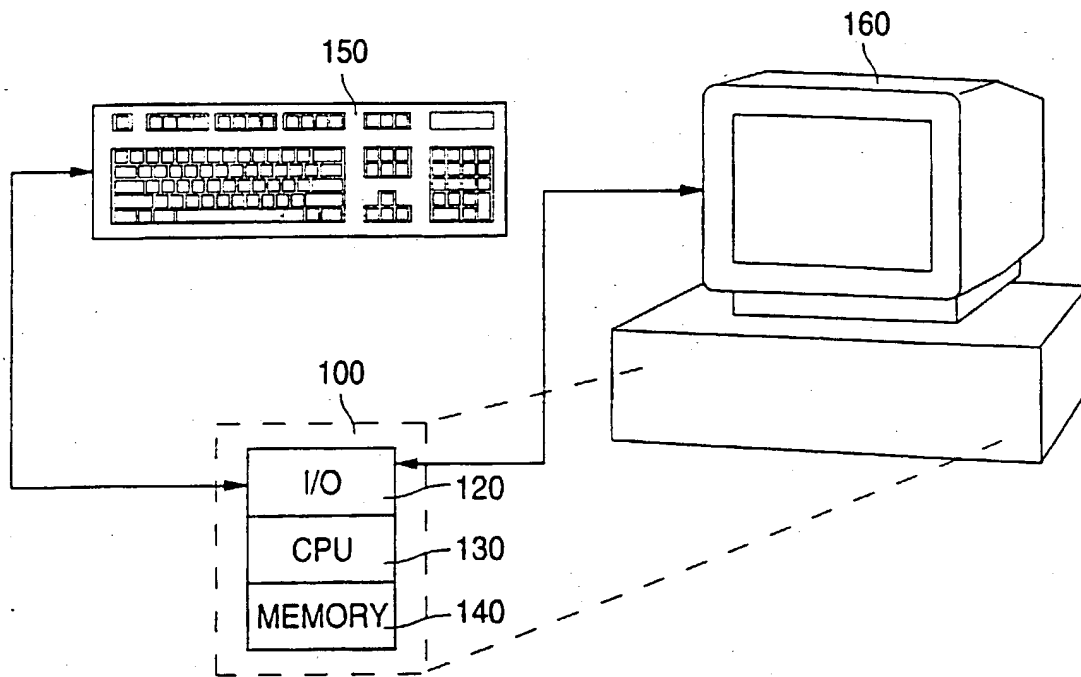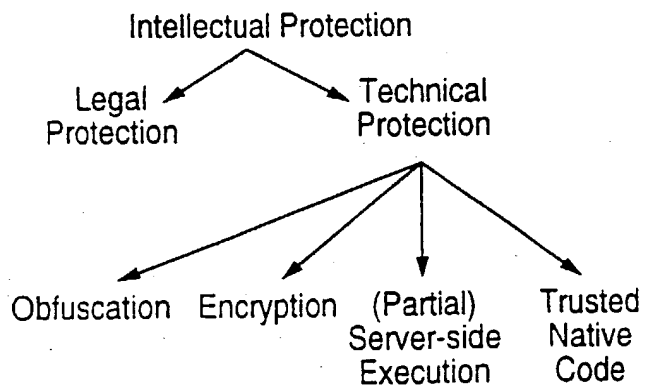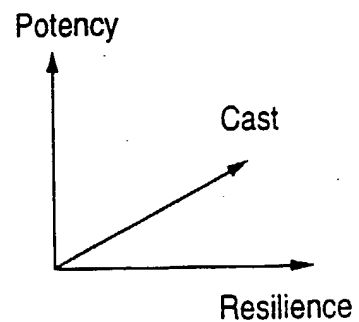behavior of the code of an application.

18.   The apparatus of Claim 13, further
15   comprising:

            means for deobfuscating the code, the
         deobfuscating the code comprising removing any
         obfuscations from the obfuscated code of an
         application by use of slicing, partial evaluation,
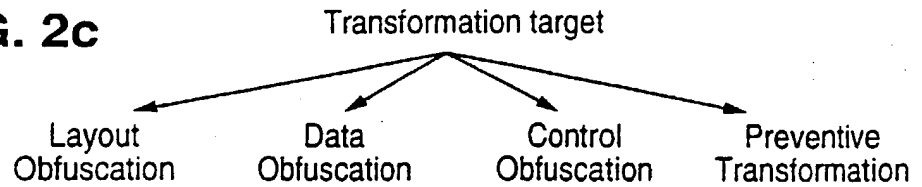20   dataflow analysis, or statistical analysis.
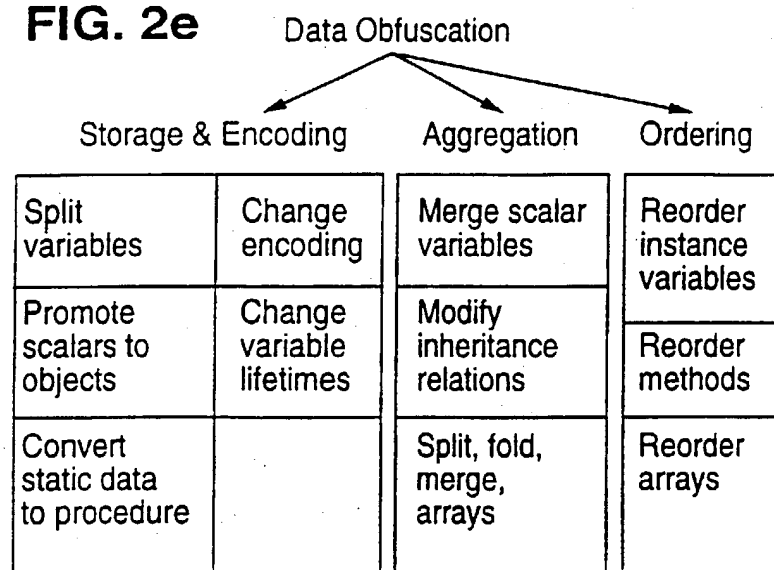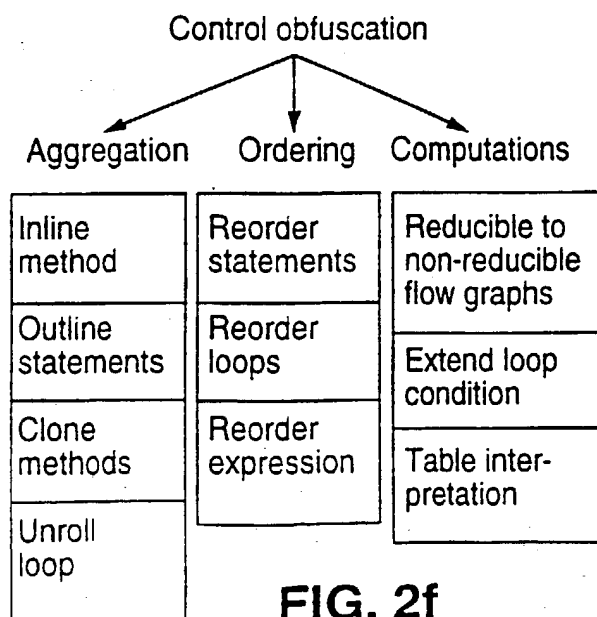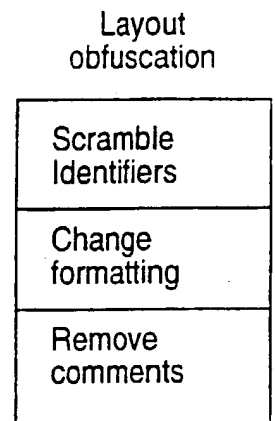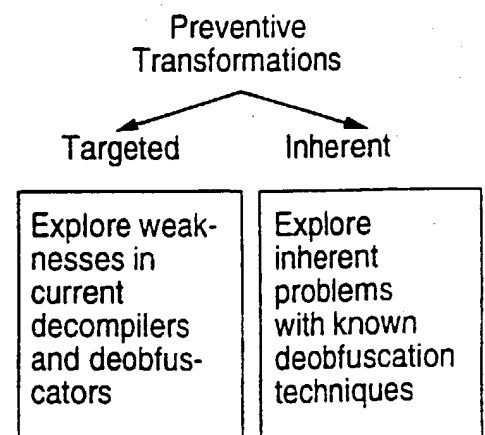

19.   The apparatus of Claim 13, wherein the code
comprises Java™ bytecode.

25   20.   The apparatus of Claim 13, wherein the
transformation provides a data obfuscation, a control
obfuscation, or a preventive obfuscation.

FIG. 1

FIG. 2a

Intellectual Protection

Legal Protection ← → Technical Protection

Technical Protection →
Obfuscation   Encryption   (Partial) Server-side Execution   Trusted Native Code

FIG. 2b

Potency

Cast

Resilience

FIG. 2c

Transformation target

- Layout Obfuscation
- Data Obfuscation
- Control Obfuscation
- Preventive Transformation

FIG. 2e

Data Obfuscation

Storage & Encoding   Aggregation   Ordering

Storage & Encoding		Aggregation	Ordering
Split variables	Change encoding	Merge scalar variables	Reorder instance variables
Promote scalars to objects	Change variable lifetimes	Modify inheritance relations	Reorder methods
Convert static data to procedure		Split, fold, merge, arrays	Reorder arrays

FIG. 2d

Layout obfuscation

Scramble Identifiers
Change formatting
Remove comments

Control obfuscation

Aggregation   Ordering   Computations

Aggregation	Ordering	Computations
Inline method	Reorder statements	Reducible to non-reducible flow graphs
Outline statements	Reorder loops	Extend loop condition
Clone methods	Reorder expression	Table interpretation
Unroll loop		

FIG. 2f

Preventive Transformations

Targeted   Inherent

Targeted	Inherent
Explore weaknesses in current decompilers and deobfuscators	Explore inherent problems with known deobfuscation techniques

FIG. 2g

FIG. 3a

FIG. 3b

FIG. 4a

FIG. 4b

FIG. 5

FIG. 6

METRIC	METRIC NAME	CITATION
μ_1	Program Length	Halstead
	$E(P)$ increases with the number of operators and operands in P.	
μ_2	Cyclomatic Complexity	McCabe
	$E(F)$ increases with the number of predicates in F.	
μ_3	Nesting Complexity	Harrison
	$E(F)$ increases with the nesting level of conditionals in F.	
μ_4	Data Flow Complexity	Oviedo
	$E(F)$ increases with the number of inter-basic block variable references in F.	
μ_5	Fan-in/out Complexity	Henry
	$E(F)$ increases with the number of formal parameters to F, and with the number of global data structures read or updated by F.	
μ_6	Data Structure Complexity	Munson
	$E(P)$ increases with the complexity of the static data structures declared in P. The complexity of a scalar variable is constant. The complexity of an array increases with the number of dimensions and with the complexity of the element type. The complexity of a record increases with the number and complexity of its fields.	
μ_7	OO Metric	Chidamber
	$E(C)$ increases with (μ_7^a) the number of methods in C_1 (μ_7^b) the depth (distance from the root) of C in the inheritance tree, (μ_7^c) the number of direct subclasses of C_1 (μ_7^d) the number of other classes to which C is coupled*, (μ_7^g) the number of methods that can be executed in response to a message sent to an object of C_1 (μ_7^f) the degree to which C's methods do not reference the same set of instance variables. Note: μ_7^f measures cohesion; i.e., how strongly related the elements of a module are.	

*Two classes are coupled if one uses the methods or instance variables of the other.

FIG. 7

Programmer
effort

Inter-process	full	full
Inter-procedural	strong	full
Global	weak	strong
Local	trivial	weak

Deobfus-cater effort

Poly time   Exp time

trivial   weak   strong   full   one-way

low resilience

high resilience

FIG. 8a
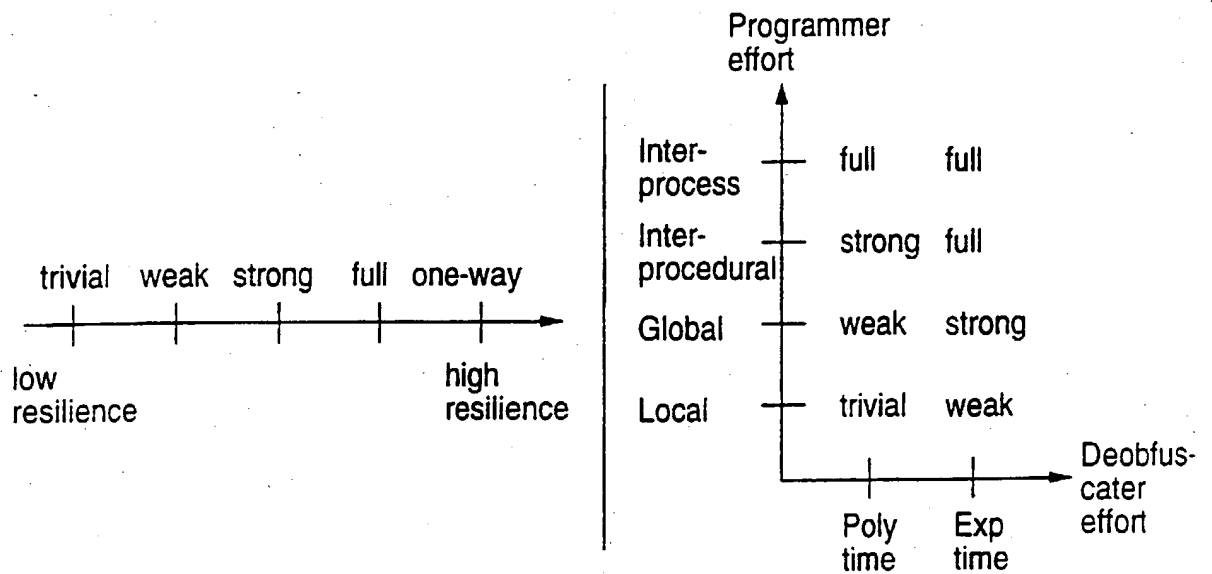
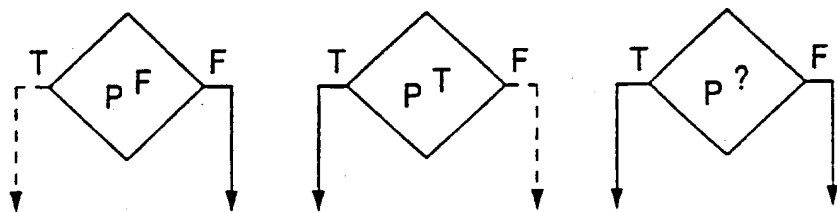**FIG. 8b**

FIG. 9

```
{    int v, a=5; b=6;
     v=11= a + b;
     if (b > 6) T ...
     if (random (1,5) < 0) F...
}
```

FIG. 10a

```
{    int v, a=5; b=6;
     if (...) ...
        ⋮     (b is unchanged)
     if (b < 7) T a++1
     v=11= (a > 5) 7v=b=b; v=b
}
```
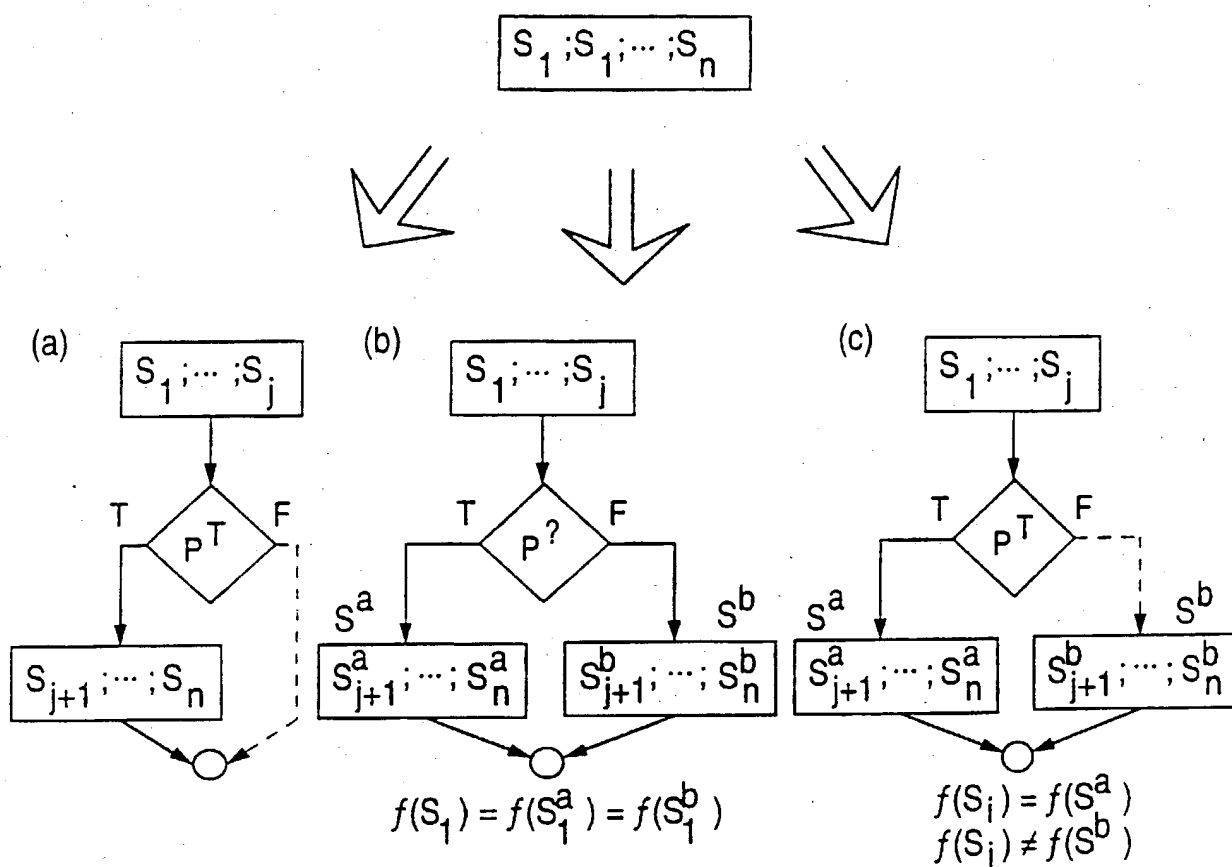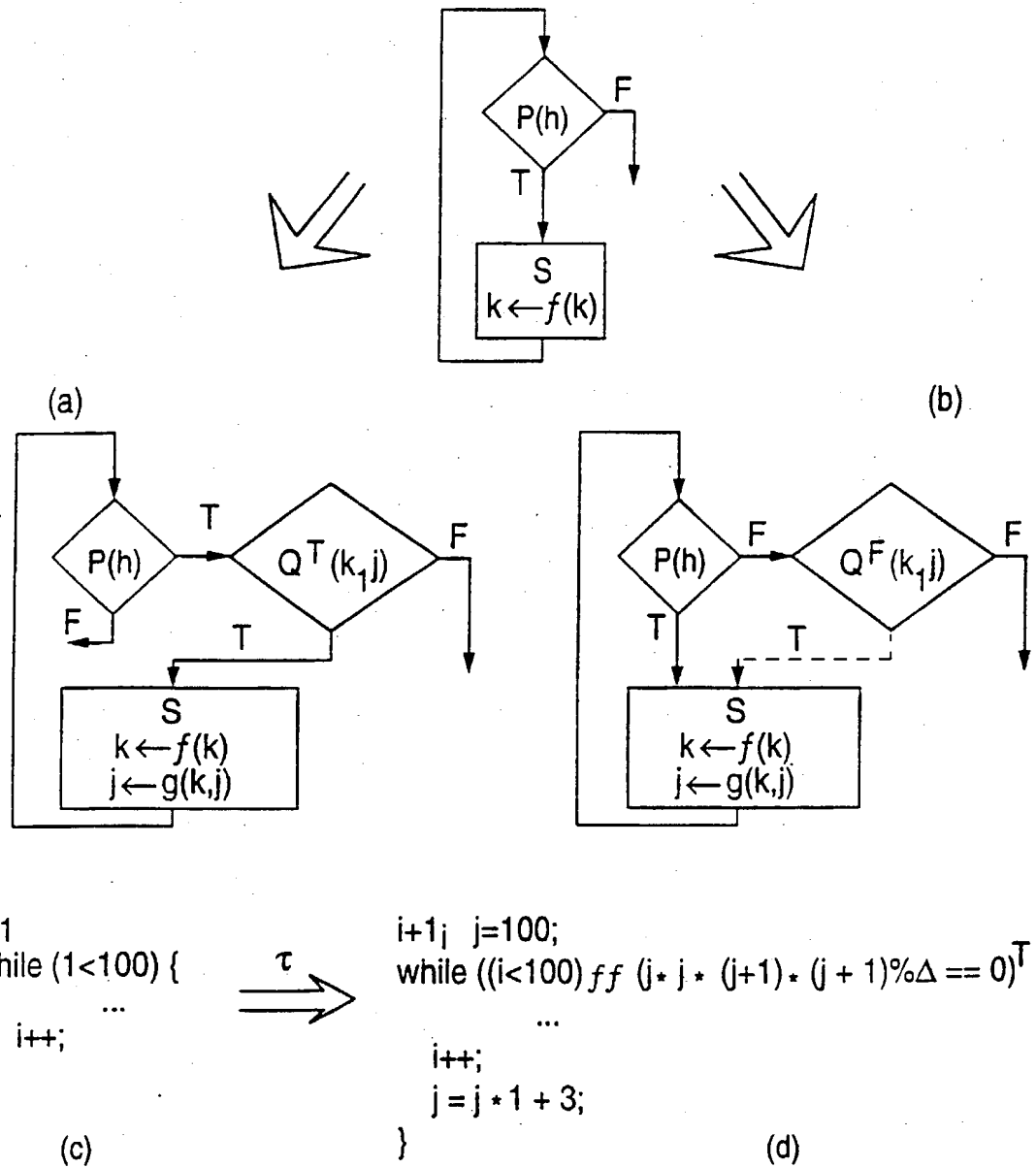
FIG. 10b

FIG. 11

(a)

(b)

(c)

(d)

$$i=1$$
while (1<100) {
    ...
    i++;
}

$\tau \Longrightarrow$

$$i+1_j \quad j=100;$$
while (((i<100) ff (j * j * (j+1) * (j + 1)%Δ == 0)^T) {
    ...
    i++;
    j = j * 1 + 3;
}

FIG. 12

11/27

FIG. 13

FIG. 14

FIG. 15

P's code

P_1
P_2
...
P_k

Q's code

Q_1
Q_2
...
Q_l

call m.P()

call n.Q()

Inline

P_1
P_2
...
P_k

Q_1
Q_2
...
Q_l

Outline

P_1
P_2
...
P_{k-1}

call R

Q_4
...
Q_l

R's code

P_k
Q_1
Q_2
Q_3

FIG. 16

call m.P()

Inline

m.type = class1

T   F

code for
class1::P

m.type = class2

T   F

code for
class0::P

FIG. 17

class C {
    method M1 (T1 a) {
      $S_1^{M1}; ... S_k^{M1};$
    }
    method M2 (T1 b; T2 c) {
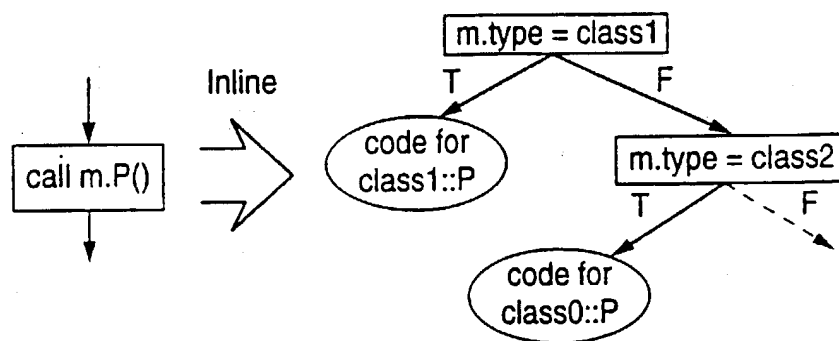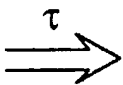 $S_1^{k1}; ... S_m^{k2};$
    }
}

{ C x=new C;
   x.M1(a); x.M2(b, c); }

$\tau \Longrightarrow$

class C' {
    method M (Ti a; T2 c; int V) {
      if (V==p) $\{S_1^{M1}; ... S_k^{M1};\}$
      else $\{S_1^{M1}; ... S_m^{k2};\}$
    }
}

{ C' x=new C';
   x.M(a, c, $V^{=p}$);
   x.M(b, c, $V^{=g}$); }

FIG. 18
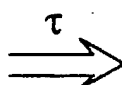
class C {
    method m (int x)
      $\{S_1 ... S_k\}$
}

{ C x = new C;
   x.m(8); ... x.m(7);
}

$\tau \Longrightarrow$

class C1 {
    method m (int x)
      $\{S_1^a; ... S_n^a\}$
    method m1 (int x)
      $\{S_1^c; ... S_n^c\}$
}
class 02 inherits C1 {
    method M (int x)
      $\{S_1^b ... S_k^b\}$
}

{ C1 x ;
   if (P7) x=new C1 else x=new C2;
   x.m(5); ...; x.m1(7);
}

FIG. 19

FIG. 20a

```
for (i=1,i<=n,i++)
   for (j=1, j<=n,j++)
      a[1,j]=b[j,i]
```
$\xrightarrow{\tau}$
```
for(I=1, I<=n, I+=64)
   for (J=1,J<=n,J+=64)
      for (i=I,i<=min(I+63,n),i++)
         for (j_J,j<=min(J+65,n),j++)
            a[i,j]=b[j,i]
```

FIG. 20b

```
for(i=2,i<(n-1),i++)
   a[i] +=a[1-i]==[i+1]
```
$\xrightarrow{\tau}$
```
for (i=2,i<(n-2),i+=2) {
   a[i] += n[i-1]=a[i+1];
   a[i+1] += a[i]=a[i+2];
};
if (((n-2) % 2) == 1)
   x[n-1] += a[n-2]=a[n]
```

FIG. 20c

```
for(i=1,i<n,i++) {
   a[i] += c;
   x[i+1]=d+x[i+1]=a[i]
}
```
$\xrightarrow{\tau}$
```
for (i=1,i<n,i++)
   a[i] += c;
for (i=1, i<n, i++)
   x[i+i] <d+x[i+1]=a[i]
```

FIG. 21a

g(V)		$f(p,q)$	
p	q	V	2p + q
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

FIG. 21b

		p	
VAL[p,q]		0	1
q	0	0	1
	1	1	0

FIG. 21c

		A			
AND[A,B]		0	1	2	3
	0	3	0	0	0
B	1	3	1	2	3
	2	0	2	1	3
	3	3	0	0	3

FIG. 21d

		A			
OR[A,B]		0	1	2	3
	0	3	1	2	3
B	1	1	1	2	2
	2	2	2	1	1
	3	0	1	2	0

```
(1)    bool A,B,C;
(2)    A = True;
(3)    B = False;
(4)    C = False;
(5)    C = A & B;
(6)    C = A & B;
(7)    C = A | B;
(8)    if (A) ...;
(9)    if (B) ...;
(10)   if (C) ...;
```
$\xrightarrow{T}$
```
(1')   short a1,a2,b1,b2,c1,c2;
(2')   a1=0; a2=1;
(3')   b1=0; b2=0;
(4')   c1=1; c2=1;
(5')   x=AND[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(6')   c1=(a1 ^ a2) & (b1 ^ b2); c2=0
(7')   x=OR[2*a1+a2,2*b1+b2]; c1=x/2; c2=x%2;
(8')   x=2*a1+a2; if ((x==1) || (x==2) ...;
(9')   if (b1 ^ b2) ...;
(10')  if (VAL[c1,c2]) ...;
```
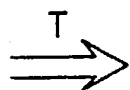
FIG. 21e

```
String G (int n) {
    int i=0,k;
    String B;
    while (i) {
        L1:  if (n==1) {S[i++]="A";k=0;goto L6};
        L2:  if (n==2) {S[i++]="B";k= -2 ;goto L6};
        L3:  if (n==3) {S[i++]="C";goto L8};
        L4:  if (n==4) {S[i++]="K";goto L9};
        L5:  if (n==5) {S[i++]="C";goto L11};
             if (n>12) goto L1;
        L6:  if (k++<=2) {S[i++]="A";goto L6} else goto L8;
        L8:  return S;
        L9:  S[i++]="C"; goto L10;
        L10: S[i++]="B"; goto L8;
        L11: S[i++]="C"; goto L12;
        L12: goto L10;
    }
}
```

FIG. 22

FIG. 23a

$$Z(X + r,Y) = 2^{32} \cdot Y + (r + X) = Z(X,Y) + r$$
$$Z(X,Y + r) = 2^{32} \cdot (Y + r) + X = Z(X,Y) + r \cdot 2^{32}$$
$$Z(X \cdot r,Y) = 2^{32} \cdot Y + X + r = Z(X,Y) + (r - 1) \cdot X$$
$$Z(X,Y \cdot r) = 2^{32} \cdot Y \cdot r + X = Z(X,Y) + (r - 1) \cdot 2^{32} \cdot Y$$

FIG. 23b

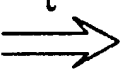|		
(1) int X=45, Y=95;		(1') long Z=167759066119551045;
(2) X += 5;	τ	(2') Z += 5;
(3) Y += 11;	$\Longrightarrow$	(3') Z += 47244640256;
(4) X *= c;		(4') z += (c-1)* (Z & 4294967295);
(5) Y *= d;		(5') Z += (d-1)* (Z & 18446744069414584320);

```
(1)   int A[9];                              (1')   int A1[4],A2[4];
(2)   A[i] = ...;                            (2')   if ((i%2)==0) A1[i/2] =...
                                                    else A2[i/2]=...;
         ...
         ...
(3)   int B[8],C[19];                        (3')   int BC[20];
(4)   B[i] = ...;                            (4')   BC[3*i] = ...;
(5)   C[i] = ...;                            (5')   BC[i/2*3+1+i%2] = ...;
                                             (6')   int D1[1,4];
         ...                                 (7')   for(j=0;j<=1;j++)
(6)   int D[9]                                          for(k=0;k<=4;k++)
(7)   for(i=o;i<=B;i++)                                     if (k==4)
          D[i]=2 * D[i+1];                                     D1[j,k]=2*D1[j+1,0];
                                                           else
         ...                                                   D1[j,k]=2*D1[j,k+1];
         ...
                                                    ...
         ...                                 (8')   int E1[8]
(8)   int E[2,2];                            (9')   for(i=0;1<=8;i++)
(9)   for(i=Q;i<=2;i ++)                                swap(E[i], E[3=(i%3)+i/3]);
          for(j=0;i<=2;i++)
              swap(E(i,j], E[j,i]);
```

<div align="center">τ ⟹</div>

FIG. 24a

FIG. 24b

FIG. 25a

FIG. 25b

$$C_1 = C_2 \oplus (V_1, M_1)$$
$$C_3 = C_3 \oplus (V_2, M_3)$$
$$C_2 = (V_2, M_2)$$

$$V_2 = V_2 \cap V_2$$
$$V_1 = V_1 - V_3$$
$$V_3 = V_2 - V_3$$

FIG. 25c

$$V = V_1 \oplus V_2$$
$$M = M_1 \oplus M_3$$

FIG. 25d

```
Node g, h;
method P(...,Node f) {
  / * 1 */ g = g.Move();
           h = h.Move();
  / * 2 */ h = h.Insert(new Node);
           ⋮
  / * 3 */ x.R(...,f.Move());
           ⋮
  / * 4 */ if (f==g) ? ...
           
  / * 5 */ if (g==h) F ...
           ⋮
  / * 6 */ f.Token=False;
           g.Token=True;
           
  / * 7 */ if (f.Token)? ...
           ⋮
  / * 8 */ f.Token=True;
           h.Token=False;
           
  / * 9 */ if (f.Token) T ...
}
```

FIG. 26

Input

```
program M;
...
if (PT) ...
...
end M.
```

```
program M';
...
if (True) ...
...
end M'.
```

Output   Output'

Identical?

FIG. 30

```
thread S {              thread T {              int X, Y;
  int R;                  int R;                 const C = sqrt(maxint)/10;
  while (1)  {            while (i)  {            main () {
    R = random(1,C);        R = random(1,C);        S.run(); T.run();
    X = R*R;                X = 7*R*R;
    sleep(3);               sleep(2);              ...
  }                         X *= X;               if ((Y - 1) == X) F <= [P]
}                           sleep(5);
                          }                        ...
                        }                        }
```
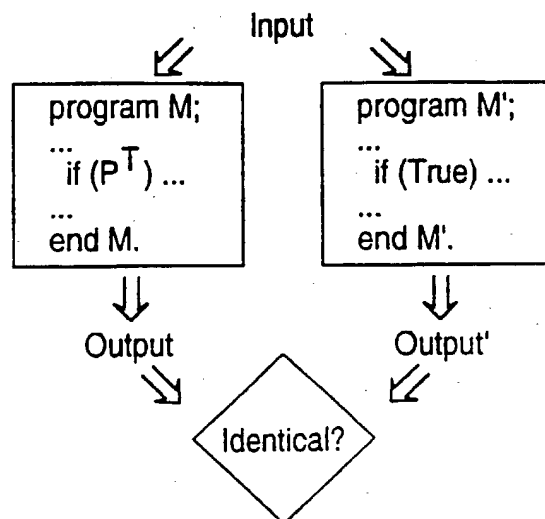
FIG. 27

$$
\begin{array}{l}
\{ \\
\quad \text{if } (P^7) \\
\quad\quad \boxed{S_1} \\
\quad \text{else} \\
\quad\quad S_1; \\
\quad \text{if } (Q^T) \\
\quad\quad \boxed{S_2} \\
\quad \text{else} \\
\quad\quad S_2^{bug} \\
\quad \text{if } (R^F) \\
\quad\quad S_3^{bug} \\
\quad\quad \boxed{S_3} \\
\}
\end{array}
$$

{
if (P⁷)
[S₁]
else
S₁;
if (Q^T)
[S₂]
else
S₂^bug
if (R^F)
S₃^bug
[S₃]
}

(a) ⟹ (b) ⟹ (c) ⟹ (d) ⟹

{
[S₁] ;
[S₂] ;
[S₃] ;
}

{
if (P⁷)
[S₁]
else
S₁;
if (True)
[S₂]
else
S₂^bug
if (False)
S₃^bug
[S₃]
}

{
[S₁] ;
if (P⁷)
else
;
if (True)
[S₂]
else
S₂^bug
if (False)
S₃^bug
[S₃]
}

{
[S₁] ;
[S₂] ;
[S₃] ;
}
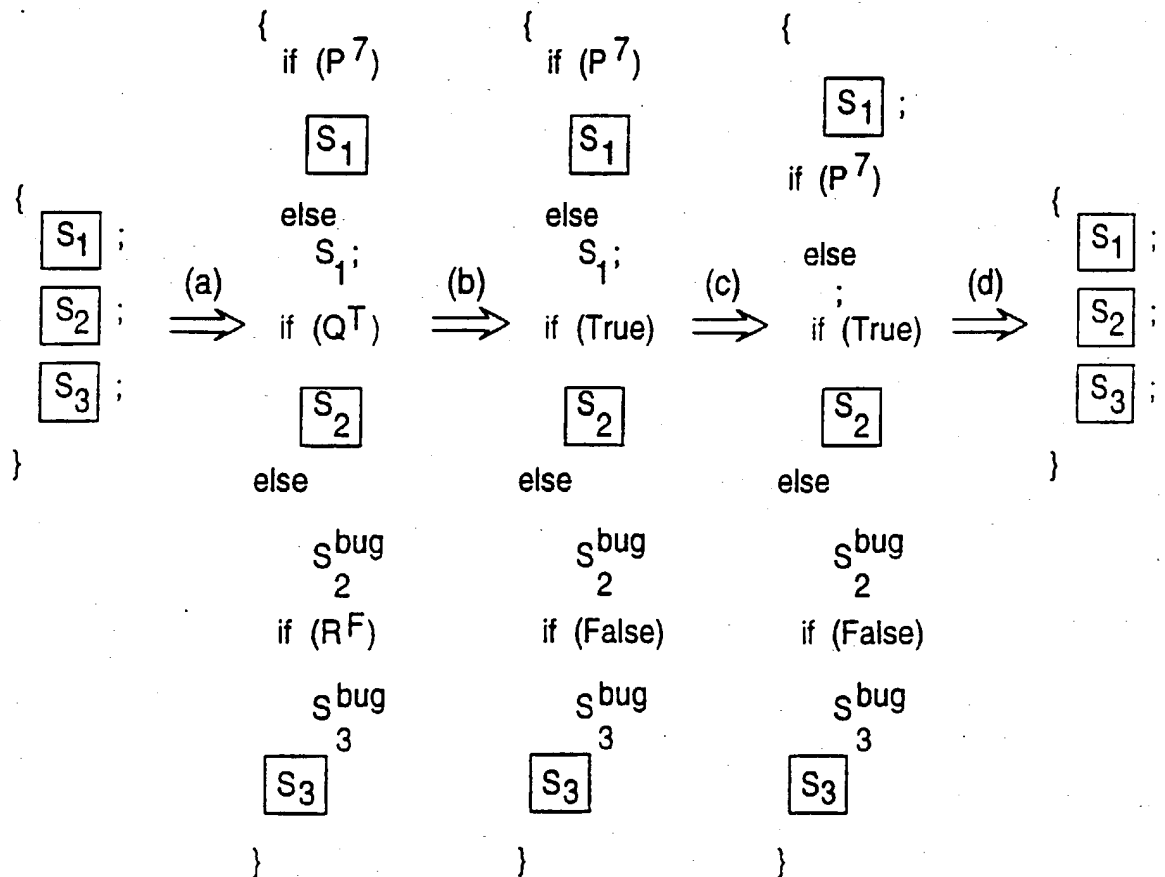
FIG. 28
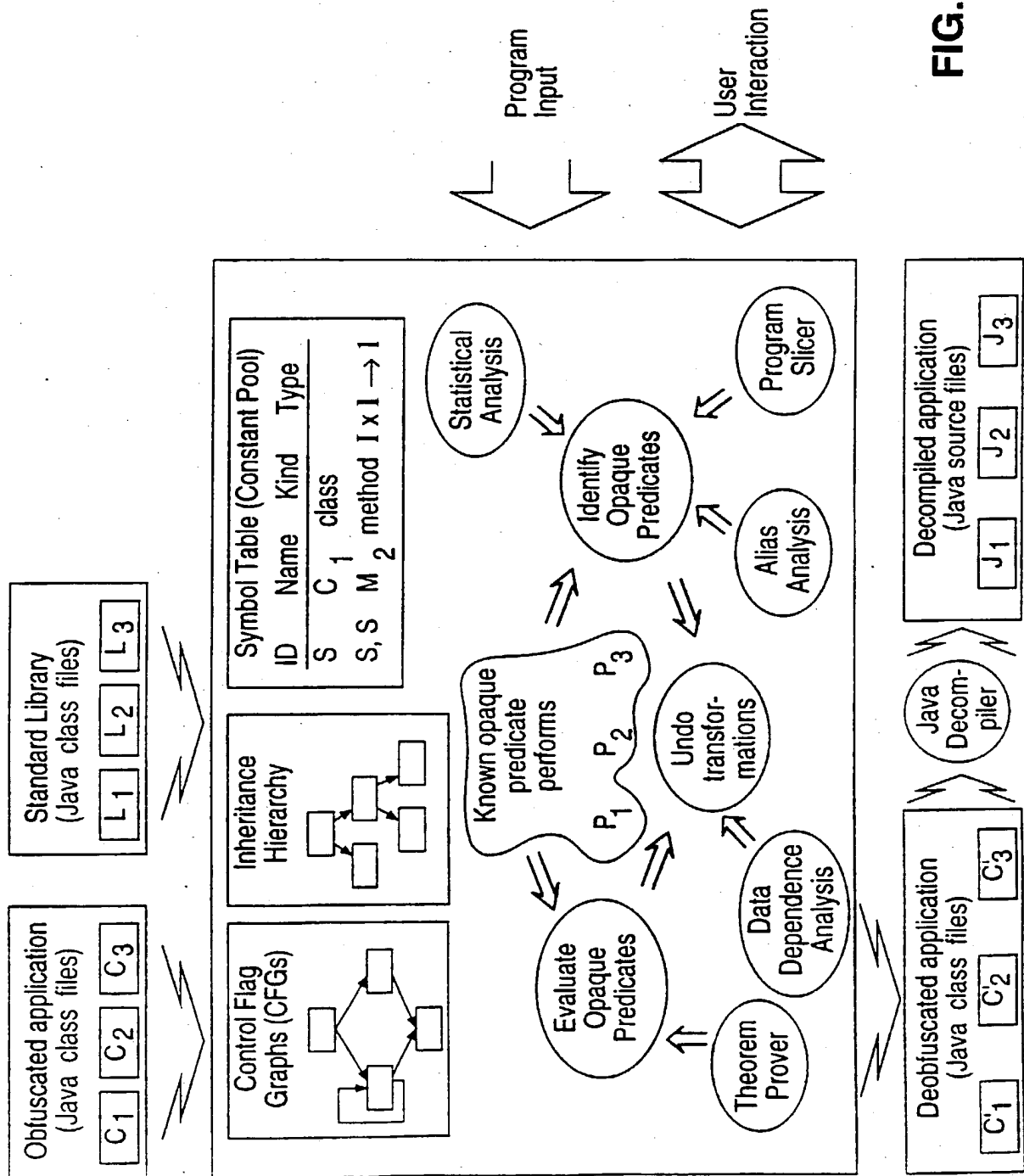
FIG. 29

TARGET	OPERATION	TRANSFORMATION	POTENCY	QUALITY RESILIENCE	COST	METRICS	SECTION
Layout		Scramble Identifiers	medium	one-way	free		5.5
		Change Formatting	low	one-way	free		5.5
		Remove Comments	high	one-way	free		5.5
Control	Computations	Insert Dead or Irrelevant Code	Depends on the quality of the opaque predicate and the nesting depth at which the construct is inserted.			μ1, μ2, μ3	6.2.1
		Extend Loop Condition				μ1, μ2, μ3	6.2.2
		Reducible to Non-Reducible				μ1, μ2, μ3	6.2.3
		Add Redundant Operands				μ1	6.2.6
		Remove Programming Idioms	medium	strong	†	μ1	6.2.4
		Table Interpretation	high	strong	costly	μ1	6.2.5
	Aggregation	Inline Method	medium	one-way	free	μ1	6.3.1
		Outline Statements	medium	strong	free	μ1	6.3.1
		Interleave Methods	Depends on the quality of the opaque predicate.			μ1, μ2, μ3	6.3.2
		Clone Methods	low	weak	free	μ1, μ7	6.3.3
		Block Loop	low	weak	free	μ1, μ2	6.3.4
		Unroll Loop	low	weak	cheap	μ1	6.3.4
		Loop Fission	low	weak	free	μ1, μ2	6.3.4
	Ordering	Reorder Statements	low	one-way	free		6.4
		Reorder Loops	low	one-way	free		6.4
		Reorder Expression	low	one-way	free		6.4

TARGET	OBFUSCATION OPERATION	TRANSFORMATION	QUALITY POTENCY	RESILIENCE	COST	METRICS	SECTION
Data	Storage & Encoding	Change Encoding	Depends on the complexity of the encoding function.			μ1	7.1.1
		Promote Scalar to Object	low	strong	free		7.1.2
		Change Variable Lifetime	low	strong	free	μ4	7.1.2
		Split Variable	Depends on the number of variables into which the original variable is split.			μ1	7.1.3
		Convert Static to Procedural Data	Depends on the complexity of the generated function.			μ1,μ2	7.1.4
		Merge Scalar Variables	low	weak	free	μ1	7.2.1
	Aggregation	Factor Class	medium	†	free	μ1, μ7 b,c,e	7.2.3
		Insert Bogus Class	medium	†	free	μ1, μ7 b,c	7.2.3
		Refactor Class	medium	†	free	μ1, μ7 b,c,e	7.2.3
		Split Array	†	weak	free	μ1, μ2, μ6	7.2.2
		Merge Arrays	†	weak	free	μ1, μ3	7.2.2
		Fold Array	†	weak	cheap	μ1,μ2,μ3,μ6	7.2.2
		Flatten Array	†	weak	free		7.2.2
	Ordering	Reorder Methods & Instance Variables	low	one-way	free		7.3
		Reorder Arrays	low	weak	free		7.3

FIG. 31a-2

OBFUSCATION			QUALITY				
TARGET OPERATION	TRANSFORMATION	POTENCY	RESILIENCE	COST	METRICS	SECTION
Preventive Targeted	HoseMocha	low	trivial	free	$\mu 1$	9
	Add Aliased Formals to Prevent Slicing	medium	strong	free	$\mu 1, \mu 5$	9.4
Inherent	Add Variable Dependencies to Prevent Slicing	Depends on the quality of the opaque predicate.			$\mu 1$	9.4
	Add Bogus Data Dependencies	medium	weak	cheap	$\mu 1$	9.1.1
	Use Opaque Predicates with Side-Effects	medium	weak	free	$\mu 1$	9.5
	Make Opaque Predicates using Difficult Theorems	†	†	†	$\mu 1$	9.5

FIG. 31b

OPAQUE CONSTRUCT	RESILIENCE	QUALITY COST	SECTION
Created from calls to library functions.	trivial	Depends on the cost of the library function	6.1.1
Created from local (intra-basic block) information.	trivial	free ... cheap	6.1.1
Created from global (inter-basic block) information.	weak	free ... cheap	6.1.1
Created from inter-procedural and aliasing information.	full	cheap ... costly	8.1
Created from process interaction and scheduling	full	cheap ... costly	8.2

FIG. 32

INTERNATIONAL SEARCH REPORT

A. CLASSIFICATION OF SUBJECT MATTER
IPC 6   G06F9/44   G06F1/00

According to International Patent Classification (IPC) or to both national classification and IPC

B. FIELDS SEARCHED

Minimum documentation searched (classification system followed by classification symbols)
IPC 6   G06F

Documentation searched other than minimum documentation to the extent that such documents are included in the fields searched

Electronic data base consulted during the international search (name of data base and, where practical, search terms used)

C. DOCUMENTS CONSIDERED TO BE RELEVANT

Category *	Citation of document, with indication, where appropriate, of the relevant passages	Relevant to claim No.
A	WO 97 04394 A (DRAKE CHRISTOPHER NATHAN) 6 February 1997 see page 3, line 25 – page 4, line 10 see page 5, line 25 – page 6, line 8	1-20
A	COHEN F B: "OPERATING SYSTEM PROTECTION THROUGH PROGRAM EVOLUTION" COMPUTERS & SECURITY INTERNATIONAL JOURNAL DEVOTED TO THE STUDY OF TECHNICAL AND FINANCIAL ASPECTS OF COMPUTER SECURITY, vol. 12, no. 6, 1 October 1993, pages 565-584, XP000415701 see the whole document	1-20
P,A	WO 97 33216 A (NORTHERN TELECOM LTD) 12 September 1997 see page 8, line 13 – page 9, line 33	1-20

☐ Further documents are listed in the continuation of box C.   ☒ Patent family members are listed in annex.

* Special categories of cited documents :

"A" document defining the general state of the art which is not considered to be of particular relevance

"E" earlier document but published on or after the international filing date

"L" document which may throw doubts on priority claim(s) or which is cited to establish the publication date of another citation or other special reason (as specified)

"O" document referring to an oral disclosure, use, exhibition or other means

"P" document published prior to the international filing date but later than the priority date claimed

"T" later document published after the international filing date or priority date and not in conflict with the application but cited to understand the principle or theory underlying the invention

"X" document of particular relevance; the claimed invention cannot be considered novel or cannot be considered to involve an inventive step when the document is taken alone

"Y" document of particular relevance; the claimed invention cannot be considered to involve an inventive step when the document is combined with one or more other such documents, such combination being obvious to a person skilled in the art.

"&" document member of the same patent family

Date of the actual completion of the international search	Date of mailing of the international search report
15 September 1998	22/09/1998

Name and mailing address of the ISA European Patent Office, P.B. 5818 Patentlaan 2 NL - 2280 HV Rijswijk Tel. (+31-70) 340-2040. Tx. 31 651 epo nl. Fax: (+31-70) 340-3016	Authorized officer Brandt, J

Form PCT/ISA/210 (second sheet) (July 1992)

Patent document cited in search report		Publication date	Patent family member(s)		Publication date
WO 9704394	A	06-02-1997	AU	5945796 A	23-01-1997
WO 9733216	A	12-09-1997	US	5748741 A	05-05-1998